

Spring 2003

# Towards more efficient solution of conditional constraint satisfaction problems

Mihaela Sabin

*University of New Hampshire, Durham*

Follow this and additional works at: <https://scholars.unh.edu/dissertation>

---

## Recommended Citation

Sabin, Mihaela, "Towards more efficient solution of conditional constraint satisfaction problems" (2003). *Doctoral Dissertations*. 133.  
<https://scholars.unh.edu/dissertation/133>

This Dissertation is brought to you for free and open access by the Student Scholarship at University of New Hampshire Scholars' Repository. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of University of New Hampshire Scholars' Repository. For more information, please contact [nicole.hentz@unh.edu](mailto:nicole.hentz@unh.edu).

## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
800-521-0600

UMI<sup>®</sup>



**TOWARDS MORE EFFICIENT SOLUTION OF  
CONDITIONAL CONSTRAINT SATISFACTION  
PROBLEMS**

BY

**MIHAELA SABIN**

M.S. in Computer Science, Bucharest Polytechnic Institute, ROMANIA, 1984

DISSERTATION

Submitted to the University of New Hampshire  
in Partial Fulfillment of  
the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

May 2003

UMI Number: 3083738

Copyright 2003 by  
Sabin, Mihaela

All rights reserved.

UMI<sup>®</sup>

---

UMI Microform 3083738

Copyright 2003 by ProQuest Information and Learning Company.  
All rights reserved. This microform edition is protected against  
unauthorized copying under Title 17, United States Code.

---

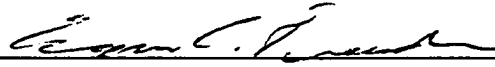
ProQuest Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

ALL RIGHTS RESERVED

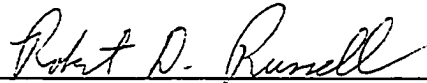
©2003

MIHAELA SABIN

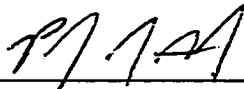
This dissertation has been examined and approved.



Dissertation Director, Eugene C. Freuder  
Professor of Computer Science, Ph.D.



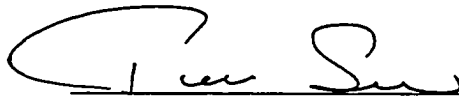
Robert D. Russell  
Associate Professor of Computer Science, Ph.D.



Philip J. Hatcher  
Professor of Computer Science, Ph.D.



Gerard Verfaillie  
Habilitation à Diriger les Recherches, Ph.D.



Paul Snow  
Consultant in Statistics and Databases, Ph.D.

5/6/03

Date



## DEDICATION

*To my parents, Maria and Gheorghe Tanase.*

## ACKNOWLEDGMENTS

**Institutional.** The research presented in this thesis has received support from the University of New Hampshire Graduate School travel grants (1996, 1997, 1999), from the National Science Foundation under Grant No. IRI-9504316, from Trilogy Software, Inc., and from Science Foundation Ireland under Grant 00/PI.1/C075.

**Personal.** Before I started working on this document, I had contemplated the uplifting moment of writing the acknowledgment page. I had imagined a time when only the printing production would separate myself from a thesis-free person. It would also bring, I was thinking, the paradoxical joy of reflecting on a task that inherently would reach agony and ecstasy, many times, with no guarantee of which of the two would be more often visited.

I am living that very moment with immense gratitude for the many people who inspired me and offered guidance and support. I would not be writing these lines had it not been for my dissertation advisor, Dr. Eugene C. Freuder. He has introduced me to the field of constraint satisfaction and convinced me about its relevance, taught me how to conduct research, and showed unconstrained patience with every single attempt I made to improve my technical writing.

I am grateful to the dissertation committee members, Dr. Robert D. Russell, Dr. Philip J. Hatcher, Dr. Gérard Verfaillie, and Dr. Paul Snow. They have waited agelessly and with confidence for this moment to happen. Their expertise and involvement have translated into the effective steps I have taken in advancing my research and, especially, in defending and writing this dissertation. Dr. Richard J. Wallace, who adapted his random CSP generator to the conditional class, made the evaluation section possible. I am thankful to him for his attentive reviews of this document.

From the proposal of this dissertation until its final submission, Dr. Esther Gelle has been my trustful companion, who knew what this journey would entail, and helped me

overcome its difficult moments. I will never forget the intensity and delight of our conversations, Esther's prompt and comprehensive responses, and her always optimistic and reassuring tone. Thank you, Esther.

The personal goes beyond my professional peers. And it is that world of personal whose sacrifice, trust, and magic have brought me here. My husband Daniel, my daughter Andreea, and my son Daniel are the doctors of this doctorate. I love you. I am very grateful to my sister Carmen and her family for making me not forget that life is family time, music and dance, and simply all seasons. I thank my friends at Rivier College for their heartfelt support. I thank all of my friends for being forgiving of my falling off the earth for the past many (secrete number) years. I am back as soon as I mark this last sentence period ;- ) Just kidding.

# TABLE OF CONTENTS

DEDICATION . . . . .	v
ACKNOWLEDGMENTS . . . . .	vi
ABSTRACT . . . . .	xiv
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.1.1 Conditional Change in Constraint Satisfaction . . . . .	1
1.1.2 Representative Applications . . . . .	3
1.1.3 Opportunities and Challenges . . . . .	4
1.2 Contributions . . . . .	5
1.3 Thesis Outline . . . . .	7
<b>2 DEFINITIONS AND TOOLS</b>	<b>9</b>
2.1 Conditional Constraint Satisfaction Problems . . . . .	9
2.1.1 Example: Simple Car Configuration . . . . .	9
2.1.2 Definitions and Notations . . . . .	14
2.2 Random Problem Generation . . . . .	20
2.2.1 Introduction . . . . .	20
2.2.2 Random Standard CSPs . . . . .	21
2.2.3 Random Conditional CSPs . . . . .	22
<b>3 SOLVING METHODS</b>	<b>27</b>
3.1 Introduction . . . . .	27
3.2 Backtrack Search . . . . .	29
3.2.1 Example . . . . .	29
3.2.2 Algorithm . . . . .	31

3.3	Forward Checking . . . . .	36
3.3.1	Example . . . . .	36
3.3.2	Algorithm . . . . .	38
3.4	Maintaining Arc and Activation Consistency . . . . .	42
3.4.1	Example . . . . .	44
3.4.2	Algorithm . . . . .	48
3.5	Summary . . . . .	58
<b>4</b>	<b>EXPERIMENTAL EVALUATION</b>	<b>59</b>
4.1	Introduction . . . . .	59
4.2	Design . . . . .	60
4.2.1	Measurements and Problem Topologies . . . . .	60
4.2.2	Experimental Studies . . . . .	62
4.3	Analysis . . . . .	64
4.3.1	Execution Time . . . . .	64
4.3.2	Counting Effort . . . . .	66
4.4	Summary . . . . .	89
<b>5</b>	<b>ON REFORMULATING CONDITIONAL CSPS</b>	<b>90</b>
5.1	Introduction . . . . .	90
5.2	Reformulation Algorithms . . . . .	92
5.2.1	Single Activations . . . . .	96
5.2.2	(Acyclic) Cluster Activations . . . . .	103
5.2.3	Activity Cycles . . . . .	113
5.3	Empirical Evaluation . . . . .	125
5.4	Summary . . . . .	129
<b>6</b>	<b>CONCLUSION</b>	<b>130</b>
	<b>REFERENCES</b>	<b>138</b>

## LIST OF TABLES

4.1	For $s_c$ in $[0.5 \dots 0.7]$ , low $s_a$ in the range $[0.1 \dots 0.3]$ and low $d_c$ of 0.3, the number of solutions has values of an order of magnitude equal to or greater only by one than the number of backtracks performed by FC and MAC algorithms. . . . .	80
4.2	For $d_c$ and $s_a$ larger than 0.6, problems have very few or no solutions, and MAC and FC perform very few backtracks. . . . .	82

## LIST OF FIGURES

2-i	A car configuration task example . . . . .	10
2-ii	Conditional CSP representation of the car configuration example . . . . .	13
3-i	Conditional CSP subproblem from the car configuration in Example 1 . . . .	28
3-ii	Backtrack search trace on the sample problem in Example 3 . . . . .	30
3-iii	Forward checking search trace on the sample problem in Example 3 . . . .	37
3-iv	New value, <i>hatchback</i> , is added to the sample problem in Example 3. $c'_{11}$ is modified to show that Package has no support for <i>hatchback</i> . Values participating in compatibility constraints have associated support counters. . . . .	45
3-v	MAC search trace of the sample problem in Example 6. $c_{10sc}$ and $c'_{11sc}$ list the support counters computed for $c_{10}$ and $c'_{11}$ constraints. . . . .	47
4-i	Execution time for running BT and FC as function of compatibility density, $d_c$ , in the range $[0.1 \dots 0.4]$ in increments of 0.02. The other three problem topology parameters are fixed: $s_c = 0.25$ , $d_a = 0.3$ , and $s_a = 0.3$ . . . . .	65
4-ii	Execution for running FC and MAC as function of activity satisfiability, $s_a$ , in the range $[0.1 \dots 0.9]$ , in 0.1 steps. Each graph corresponds to a different density of activity, $d_a$ , varied in the range $[0.1 \dots 0.9]$ in 0.1 increments. The compatibility topology is fixed: compatibility density, $d_c$ , and compatibility satisfiability, $s_c$ are set at 0.2. . . . .	67
4-iii	BT and FC performance measured by number of backtracks, compatibility checks, and condition checks done by each algorithm as functions of $s_a$ for fixed $d_c = 0.4$ , $s_c = 0.1$ , and $d_a = 0.1$ (top - both BT and FC; middle - only FC). Ratios between corresponding counters show the factor by which FC outperforms BT (bottom). . . . .	68
4-iv	Comparison between BT and FC effort measured as the number of backtracks (rows 1 and 3), ratio of number of backtracks (rows 2 and 4), and number of compatibility and condition checks (row 5). Variation of effort with compatibility density, $d_c$ , and activity satisfiability, $s_a$ . Fixed activity density, $d_a = 0.2$ (rows 1, 2, and 5) and 0.6 (rows 3 and 4). Each column corresponds to a different compatibility satisfiability value: 0.1 (left), 0.3 (middle), and 0.5 (right). . . . .	70
4-v	Comparison between BT and FC using the same effort measures as in Figure 4-iv. Variation of effort with compatibility density, $d_c$ , and activity density, $d_a$ . Fixed activity satisfiability, $s_a$ , of 0.2 (rows 1, 2, and 5) and 0.6 (rows 3 and 4). . . . .	71
4-vi	Comparison between BT and FC using effort measures as in Figure 4-iv and Figure 4-v. Variation of effort with $d_a$ and $s_a$ . Fixed compatibility density of 0.4 (rows 1, 2, and 5) and 0.8 (rows 3 and 4). . . . .	72
4-vii	Relative performance of FC and MAC measured by number of backtracks (left) and variation of number of solutions (right) as functions of $s_a$ . Algorithms are run on random problems in $(d_c, s_c, d_a) = (0.3, 0.3, 0.6)$ class. Performance and solution values are averages over sets of 10 problems for each $(d_c, s_c, d_a, s_a)$ topological point. . . . .	74

4-viii	For problems with very large solution sets (top), MAC and FC perform very similarly in terms of number of backtracks (middle). The test suite uses random problems in three $(d_c, s_c)$ classes: (0.1, 0.5), (0.2, 0.6), and (0.3, 0.9) for which $d_a$ takes on low values of 0.1, 0.2, and 0.3. . . . .	75
4-ix	Fixing $d_a$ at 0.6 and varying $s_a$ in the high range of [0.5...0.9], we select $(d_c, s_c)$ topologies for which the number of solutions is roughly the same (top). On all these problems MAC outperforms FC (rows 2 and 3). Cost effectiveness of MAC algorithm is measured by computing FC backtracks over MAC backtracks (bottom). . . . .	78
4-x	(Left) Variation of number of solutions for fixed $d_a = 0.5$ , three satisfiability levels of 0.5, 0.6, and 0.7, and variable $d_c$ in [0.3...0.8] range and $s_a$ in [0.1...0.9] range: linear scale (top) and log scale (bottom). (Right) Data sets for the number of solutions plotted by $\text{sols} - d_c - 7 - 5$ (top) and data sets over ranges of high $d_c$ in [0.6...0.8], and high $s_a$ in [0.6...0.9], for the number of solutions of all graphs categories (bottom). . . . .	80
4-xi	Variation of number of backtracks on the problem set in Figure 4-x, with each column corresponding to different satisfiability value: 0.5 (left), 0.6 (middle), and 0.7 (right). Comparison between MAC and FC performance using a log scale (top); difference in number of backtracks between FC and MAC (middle); performance factor of FC backtracks over MAC backtracks (bottom). . . . .	81
4-xii	Problems of variable conditionality: $d_a$ and $s_a$ varied in [0.1...0.9] and fixed $d_c = 0.3$ and $s_c = 0.6$ . (First row) Number of solutions on normal scale (left) and logarithmic scale (middle); significantly smaller solution sets for $d_a$ and $s_a$ in the second half of their interval (right). Variation of activity counters: condition checks (row 2), included variables (row 3), and excluded variables (row 4). Relative performance of FC and MAC is shown for each activity counter: both MAC and FC effort surfaces on normal scale (left) and logarithmic scale (middle), and ratio of MAC and FC corresponding counters (right). . . . .	85
4-xiii	Continuation of Figure 4-xii. Activity counters: redundant activations (top) and conflicting activations (bottom). MAC checks fewer redundant activity constraints, but more conflicting activity constraints. . . . .	86
4-xiv	Problems of variable densities, $d_c$ and $d_a$ , and fixed satisfiability parameters, $s_a = 0.4$ and $s_c = 0.6$ . (First row) Number of solutions on normal scale (left) and logarithmic scale (middle); largest solution sets controlled by low compatibility density, $d_c$ , of 0.3 and 0.4. Variation of activity counters and relative performance of FC and MAC are reported in the same fashion as in previous experiment in Figure 4-xii. . . . .	87
4-xv	Continuation of Figure 4-xiv. Variation of redundant (top) and conflicting (bottom) activations. MAC checks fewer redundant activity constraints and more conflicting activity constraints . . . . .	88
5-i	Simple conditional CSP example, $\mathcal{P}_1$ . . . . .	94
5-ii	Simple conditional CSP example $\mathcal{P}_2$ with $a_2$ exclusion activity constraint . . . . .	100
5-iii	Reformulation $\mathcal{P}_{R1}$ of the problem example $\mathcal{P}_1$ in Figure 5-i shows the reformulation of single activation inclusion constraints. . . . .	102

5-iv	(Left) Conditional CSP example $\mathcal{P}_3$ with cluster activations. (Right) Reformulation $\mathcal{P}_{\mathcal{K}_3}$ of $\mathcal{P}_3$ . (Bottom) Problem solution set. . . . .	104
5-v	(Top) Reformulation of a single activity constraint of inclusion. (Bottom) Incremental reformulation of an additional cluster activation from a previous reformulation. . . . .	112
5-vi	Simple conditional CSP example, $\mathcal{P}_4$ . (Left) Problem description. (Right) $\mathcal{P}_4$ 's activity graph. . . . .	115
5-vii	(Top) Incorrect reformulation of $\mathcal{P}_4$ . (Bottom) Correct reformulation of $\mathcal{P}_4$ . . . . .	115
5-viii	Simple conditional CSP example, $\mathcal{P}_5$ . (Left) Problem description. (Right) $\mathcal{P}_5$ 's activity graph. . . . .	117
5-ix	Incorrect reformulation of $\mathcal{P}_5$ . . . . .	117
5-x	Correct reformulation for $\mathcal{P}_4$ . . . . .	121
5-xi	Execution time of <i>CCSP_Mac</i> , which solves the conditional CSP, and <i>RefMAC</i> , which solves its binary null-based reformulation. The original conditional CSP has 8 variables and 6-value domains. 100 problem instances are generated in each topological class: $s_c$ in $[0.1 \dots 0.9]$ and fixed $d_c = 0.15$ , $d_a = s_a = 0.3$ . . . . .	127
5-xii	Domain size averages (left) and solution size averages (right) of the binary null-based reformulations, dom-binary-ref and sol-binary-ref plots. These reformulations are obtained from the conditional CSPs in our test suite. The domain size of the conditional CSPs is 6 (not plotted). sol-conditional plot (right) shows the solution size averages for the original conditional CSPs. . . . .	128

**ABSTRACT**

**TOWARDS MORE EFFICIENT SOLUTION OF  
CONDITIONAL CONSTRAINT SATISFACTION  
PROBLEMS**

by

MIHAELA SABIN

University of New Hampshire, May, 2003

The focus of the thesis is on improving solving constraint satisfaction problems (CSPs) that change with certain conditions. This special class of problems, which we call *conditional CSPs*, has proved very useful in modeling important applications, such product configuration and design, and distributed software diagnosis and network management. The problem conditions model choices customers make to configure a product, or they are installation settings or actual observations of a running system that is monitored for diagnosis purpose.

The key, novel contribution of this thesis are two approaches for improving solving methods and the use of random conditional CSPs to evaluate the performance of these methods. With the first approach we propose new algorithms for solving conditional CSPs. These algorithms propagate problem constraints and conditions. The second approach explores the feasibility of reformulating the problem into a standard CSP and introduces new reformulation algorithms.

The implementation results have been evaluated experimentally. The experimental design has extensive test suites of randomly generated standard and conditional CSPs for which general problem parameters, such as density and satisfiability, were varied, as well as specialized parameters that characterize the representation of problem conditions.

The significance of the work lies in the advance of problem resolution for the class of

conditional CSPs and the experimental analysis for the proposed new algorithms. The limited solving developments known in the literature of the class of conditional CSPs, a backtrack search algorithm tested on a handful of small problem examples, have been taken an important step further and aligned with efforts reported for standard and other special classes of CSPs.

# CHAPTER 1

## INTRODUCTION

The main topic of this dissertation is improving the solving of conditional CSPs. In this chapter we present the underlying motivation for pursuing this research, and list the contributions made. We conclude with an outline of the dissertation chapters.

### 1.1 Motivation

#### 1.1.1 Conditional Change in Constraint Satisfaction

There are many important and complex tasks to which constraint satisfaction has been successfully applied. Among these tasks are action planning and task scheduling, design and configuration, verification and diagnosis. The constraint satisfaction paradigm provides a natural, simple, yet generic modeling language, and employs well-established and efficient solving algorithms. At the center of the paradigm lies the concept of *constraint satisfaction problem* (CSP), defined simply by its three components: a set of variables, their associated domains of values, and a set of constraints which restrict the allowed value combinations variables can take. A solution to a CSP is a value assignment to all its variables such that all constraints are satisfied.

Important, specialized CSP classes have been defined to cope more directly with specific characteristics of various application domains. Qualifiers such as partial, optimization, dynamic, hierarchical, composite, interval, continuous, mixed, fixed-point, and others characterize CSP specializations that have been studied in the last decade. Conditional constraint satisfaction problem is another example of adapting constraint technology to better apply to diagnosis and configuration domains.

Conditional CSP extends standard CSP with a condition-based component that models dynamic changes of the problem with predefined conditions. Known as dynamic constraint satisfaction problem (DCSP), the formalism was introduced by Mittal and Falkenhainer in 1990 (Mittal & Falkenhainer 1990) to integrate classical constraint satisfaction with a special type of constraint, *activity constraint*, responsible for selecting variables that could participate in solutions. The formalism was originally motivated by synthesis tasks such as product configuration, in which not all cataloged components have to be present in every configured product.

We renamed this class of dynamic CSPs *conditional constraint satisfaction problems* (Sabin, M. & Freuder 1998) to:

- capture the nature of the control component that conditionally changes the initial model of the problem, and to
- distinguish this class of problems from another class of dynamic CSPs for which attention is focused on reusing problem solutions when the problem changes over time (Dechter & Dechter 1988), (Bessiere 1991), and (Verfaillie & Schiex 1994).

In general, conditional constraint satisfaction adds to the standard paradigm the following distinctive capabilities:

- representation of problem changes by conditioning what variables and constraints define the problem while searching for solutions,
- seamless integration of the control mechanism for dynamic model change into the problem formulation, and
- run-time selection of model components that supports user interaction or monitored observations.

In the following, we give examples of some representative applications for which conditional CSP capabilities have proved very useful.

### 1.1.2 Representative Applications

The application domain that originally motivated conditional CSP is *equipment configuration* (Frayman & Mittal 1987), (Mittal & Frayman 1989), (Mittal & Falkenhainer 1990). ILOG and Trilogy<sup>1</sup> are two examples of companies that incorporate conditional CSP features in their technologies to provide business solutions for equipment, sales, and service configuration in domains that range from aerospace and automotive to computers, electronics, and telecommunications.

In essence, a product configuration task is about configuring an extremely large number of variants, on the order of hundred of thousands, from one encompassing description. That description specifies the parts participating in all variants, along with customer selections and conditions under which certain variants can be configured. The conditions capture product assembly knowledge and promotional sales strategies. The task is to find variants that satisfy all conditions and customer requests. A successful series of Configuration Workshops, which started in 1996 as a AAAI Fall Symposium<sup>2</sup>, has captured the attention of both academia and industry every year since 1999. The IJCAI 2003 Configuration Workshop (Mailharro 2003) continues to promote a strong synergy between research and major configurator vendors. We direct the reader to the workshop proceedings for a comprehensive view on the state-of-the-art in this application domain.

Since its first formalization in 1990, the conditional constraint satisfaction paradigm has been used for modeling application problems in other domains, such as *diagnosis* of distributed software systems (Sabin, D. *et al.* 1995), (Sabin, M. & Freuder 1996), *conceptual design* of bridges (Gelle 1998), *network management* of domain name service (Sabin, M., Russell, & Freuder 1997), groupware services (Sabin, M. *et al.* 1999), and LAN configuration (Sabin, M., Russell, & Miftode 2001).

---

<sup>1</sup>©ILOG, Inc. and Trilogy are registered trademarks.

<sup>2</sup>Information available at <http://www.aaai.org/Symposia/Fall/1996/>.

### 1.1.3 Opportunities and Challenges

Despite increasing interest in the area of representing application problems as conditional CSPs, little progress has been made in the area of improving solving methods for conditional CSPs. In contrast with other CSP specializations, no standard CSP solving method, except for backtracking search (Gelle 1998), has been adapted to the conditional domain. The standard domain teaches us that enforcing local consistency, such as forward checking and maintaining arc consistency, can be embedded into backtrack search to reduce the search space (Gaschnig 1974), (Mackworth 1977), (Haralick & Elliott 1980), (Sabin, D. & Freuder 1994), (Bessiere & Regin 1996). The first topic of this thesis is the design of new algorithms for solving conditional CSPs that combine backtracking with local consistency.

The lack of specialized, direct solving methods is compounded by the fact that a benchmark test base for this type of problems is extremely limited (Soininen, Gelle, & Niemelä 1999), although very much needed in experimental evaluations of such methods. The reality of many application domains, such as configuration or diagnosis, is that either real-life problem data is not publicly available or problem examples are too simple. The opportunity of importing efficient standard algorithms, whose behavior has been extensively tested, raises new challenges for the conditional CSP class. Are there available similarly comprehensive experimental studies for evaluating conditional CSPs? What topological features make conditional CSPs hard? What metrics are suitable for evaluating the relative performance of the new methods?

A practical approach that overcomes these drawbacks and has been proved very successful for benchmarking standard solving algorithms is randomly generated CSPs (MacIntyre *et al.* 1998), (Achlioptas *et al.* 2001). Wallace extends his random standard CSP generation model (Wallace 1996) to produce random activity constraints, and uses the model to implement a random conditional CSP generator. The generator has new parameters for controlling problem activity, in addition to typical parameters for specifying problem size and topological features, such as density and satisfiability. The second topic of this thesis is to evaluate empirically the proposed algorithms using random conditional CSPs.

An alternative approach to specialized solving methods is to reformulate conditional CSPs into their standard analogs. The approach has the advantage of bringing to bear a mature constraint technology that has the means to match problem representation with the most adequate reasoning methods. A first reformulation of conditional CSP into standard CSP was mentioned by Mittal and Falkenhainer (Mittal & Falkenhainer 1990), although an exact transformation was not provided. They consider the addition of a special value, called “null”, to the domains of all variables which are not initially active. A variable instantiation with “null” indicates that the variable does not participate in the problem solution. Haselböck proposes a partial reformulation of a conditional CSP that transforms into regular constraints only activity constraints that exclude variables from solutions (Haselböck 1993).

The feasibility of obtaining a null-based CSP formulation from a conditional CSP is examined in-depth by Gelle (Gelle 1998). She proposes a reformulation algorithm for conditional CSPs whose variables are activated by single activity constraints. The case of multiple activity constraints that condition the inclusion of the same variable into the problem is recognized as not straightforward. Gelle gives the idea that multiple activations be clustered into a single activity constraint. She also warns that problems with cluster activations do not allow for an incremental reformulation: a local change of adding a new activity constraint in the original problem does not entail a local change of adding a reformulated constraint. The third topic of the dissertation is a study of the feasibility of reformulating conditional CSPs into standard CSPs, and the design of reformulation algorithms.

## 1.2 Contributions

### Topic 1: New algorithms for solving conditional CSPs

#### Contribution

In this thesis we present two original solving methods for conditional CSPs that extend local consistency methods of forward checking and maintaining arc consistency to process

the new constraints of activity. With the checking of the activity constraints, the initial set of variables that are assigned values in every solution changes with additional variables that are either included in the solution space (or made active) or explicitly excluded from it. The problem considered by the new algorithms, at any point in time during search, is given by the set of active variable. Thus, forward checking propagates the current instantiation over standard constraints to all the active, non-instantiated variables. Maintaining arc consistency uses the current instantiation to make all active variables arc consistent. Standard arc consistency, over standard, compatibility constraints, is enhanced with enforcing consistency over activity constraints. We also define a new type of local consistency, over activity constraints, and use it to further improve the efficiency of the new maintaining arc consistency algorithm.

## **Topic 2: Empirical evaluation of solving methods**

### **Contribution**

The proposed algorithms are tested in experiments covering large and topologically diverse populations of random conditional CSPs. In the experimental studies, algorithm effort is measured by timing algorithm execution, and by counting search operations specific to standard and conditional CSP solving. We show that maintaining arc consistency outperforms forward checking, which, in turn, outperforms backtracking. We observe that the improvement in performance is strongly dependent on the size of the solution space even when algorithms search for solutions of minimum size rather than for all solutions.

## **Topic 3: Reformulation feasibility and algorithms**

### **Contribution**

We present a formal definition for reformulating conditional CSPs into standard CSPs. We use the theoretical framework of null-based reformulation to develop an original algorithm of reformulation that addresses the problem of multiple activations of the same variable. We discover that multiple activations might introduce activity cycles whose transformations

add new constraints to the reformulated problem. Under a less restrictive notion of local change, that allows both local addition and removal of constraints, we develop a null-based reformulation algorithm that localizes change in the original problem. Local transformations, however, are possible only in the absence of activity cycles. A new, more general reformulation algorithm is presented that transforms conditional CSPs which have activity cycles. This algorithm has the limitation of not preserving locality of change. We evaluate experimentally the performance of solving the reformulated problem obtained from running the general reformulation algorithm. Performance results are compared with results obtained from applying the direct solving methods we developed to the original problem.

### 1.3 Thesis Outline

- Chapter 2, Definitions and Tools, presents a formal definition of the conditional CSP and reviews a model for random generation of CSPs.
- Chapter 3, Solving Methods, introduces two new solving methods for conditional CSP that use local consistency in the presence of the condition-based component that is specific to conditional CSPs. Local consistency is extended with a new type of consistency over activity constraints.
- Chapter 4, Experimental Evaluation, reports performance results of the solving methods introduced in Chapter 3. Experimental studies use a special class of random conditional CSPs. Computational cost of the studied algorithms is measured by counting representative search operations and by timing algorithm execution.
- Chapter 5, On Reformulating Conditional CSPs, proposes three new reformulation algorithms that produce standard CSP representations of conditional CSPs. Two algorithms address the difficulties with transforming multiple activations and preserving locality of change in the absence of activity cycles. The third algorithm provides a general null-based reformulation that handles activity cycles. Experimental analysis shows that solving the reformulated problem with standard CSP algorithms is less

effective than directly solving the original problem.

- Chapter 6, Conclusion and Future Work, reviews the thesis contributions and discusses open research topics these contributions entail.

## CHAPTER 2

### DEFINITIONS AND TOOLS

In this chapter we recall the theoretical framework of conditional CSP, and present Wallace’s model of generating random CSPs, standard and conditional. We start with an example of a simple product configuration task for which we develop a conditional CSP representation. The insights of the modeling exercise facilitate the introduction of a formal definition of the conditional CSP class. The description of Wallace’s model for generating random standard and conditional CSPs comes next. The problem generation tools are essential to the design and analysis of the experimental studies we conduct to measure the computational cost of the proposed solving methods.

## 2.1 Conditional Constraint Satisfaction Problems

### 2.1.1 Example: Simple Car Configuration

Before we give a formal definition of the class of conditional constraint satisfaction problems, we introduce an example of a car configuration task that can be modeled as a conditional CSP. The example is a simplified version of an example introduced by (Mittal & Falkenhainer 1990).

**Example 1.** We start with specifying a car configuration task. (Figure 2-i).

The specifications include:

- Required components, that participate in all final car configurations, such as frame and engine;
- Optional components, such as air conditioner and sunroof, that can be optionally selected according to certain configuration requirements,

*Required components and their values*

- comfort package has luxury, deluxe, and standard values
- frame has convertible, sedan, and hatch back values
- engine has small, medium, and large
- battery has small, medium, and large value

*Optional components and their values*

- sunroof has sr1 and sr2 values
- sunroof glass has tinted and not-tinted values
- sunroof opener has manual and automatic values
- air conditioner has ac1 and ac2 values

*Configuration requirements of compatibility among component values*

1. standard comfort package is not compatible with ac2 air conditioner
2. luxury comfort package is not compatible with ac1 air conditioner
3. standard comfort package is not compatible with convertible frame
4. automatic sunroof opener and ac1 air conditioner are compatible only with medium battery
5. automatic opener and ac2 air conditioner are compatible only with large battery
6. sr1 sunroof and ac2 air conditioner are not compatible with tinted glass

*Configuration requirements for selecting optional components*

1. luxury comfort package includes sunroof option
2. luxury comfort package includes air conditioner option
3. deluxe comfort package includes sunroof option
4. sr2 sunroof includes opener option
5. sr1 sunroof includes air conditioner option
6. sunroof always includes sunroof glass
7. convertible frame excludes sunroof option
8. small battery and small engine excludes air conditioner option

Figure 2-i: A car configuration task example

- Values for each component, such as convertible, sedan, and hatch back for the frame component,
- Configuration requirements of compatibility, that restrict the values of the selected components to ensure the correct functionality of the resulting configuration. For example, an automatic sunroof opener and a certain type of air conditioner work only

with a medium size battery.

- Configuration requirements for selecting optional components, that express customer specifications or additional functional requirements. A customer might request, for example, a luxury comfort package in order to include two options to the configuration: sunroof and air conditioner. Another example is a configuration requirement imposed on considering optional components, such as the functional restriction that a small battery and small engine selection exclude the air conditioner option.

Given the specified components and requirements, the task of configuration is to assign values assigned to selected components in such a way that requirements pertaining to what is selected are satisfied. An example of a valid configuration is a car whose comfort package is deluxe, has a sedan frame, a medium engine, a sr1 sunroof, ac2 air conditioner, and a tinted sunroof.

△

The car configuration example is used here to introduce the basic concepts of the conditional CSP formalism. To model the car configuration task as a conditional CSP, one should:

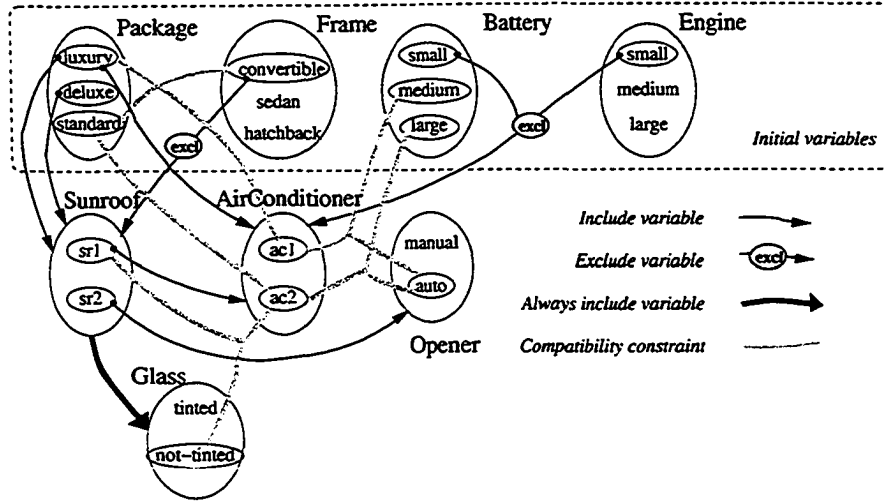
- Identify the problem's variables, which correspond to required and optional car components.
- Delimit the variables corresponding to required components, which are part of any configuration solution. We call these variables *initial* or *start variables*.
- Identify the values in each variable domain.
- Express two types of constraints to model requirements of component compatibility and selection:
  - compatibility constraints, which restrict the combinations of allowed values to the selected components, and

- activity constraints, which change the initial variable set according to certain conditions. These conditions control what optional components get selected in a configuration. Selected components correspond to *active variables*. Note that initial variables are always active since they are automatically selected in any solution.

By following these general modeling guidelines, we obtain a conditional CSP representation of the car configuration task, whose variables, domains of values, and constraints are shown in Example 2. We recall that a standard CSP can be represented as a constraint network. Variables with their domains are represented as labeled nodes that list domain values. Constraints are represented as edges or hyper-edges that connect the variables on which constraints are defined. The representation of a conditional CSP adds to the representation of the standard constraint network the activity constraints, and delimits those nodes that correspond to the initial variable subset. The activity constraints are represented as directed edges or hyper-edges that point to non-initial variable that are included or excluded from the active variable set.

**Example 2.** The conditional CSP model of the car configuration task in Example 1 has eight variables  $\{Package, Frame, Engine, Sunroof, AirConditioner, Battery, Glass, Opener\}$ , each of which has a domain of values, eight activity constraints  $\{a_1, \dots, a_8\}$ , and six compatibility constraints  $\{c_9, \dots, c_{14}\}$ . Four of the problem variables,  $\{Package, Frame, Engine, Battery\}$ , are initial variables and, therefore, active. They describe the initial problem with which the solving process starts. It does so by checking whether the combinations of values chosen for these variables comply with configuration requirements formulated as constraints.

The activity constraints change the set of active variables when certain conditions become true during search. New variables are dynamically included in the set of active variables and become candidates to problem solutions. Other new variables are dynamically



### Activity Constraints

- $$\begin{aligned}
 a_1 : \text{Package} = \text{luxury} &\xrightarrow{\text{incl}} \text{Sunroof} & a_5 : \text{Sunroof} = \text{sr1} &\xrightarrow{\text{incl}} \text{AirCond} \\
 a_2 : \text{Package} = \text{luxury} &\xrightarrow{\text{incl}} \text{AirCond} & a_6 : \text{Sunroof} &\xrightarrow{\text{incl}} \text{Glass} \\
 a_3 : \text{Package} = \text{deluxe} &\xrightarrow{\text{incl}} \text{Sunroof} & a_7 : \text{Frame} = \text{convertible} &\xrightarrow{\text{excl}} \text{Sunroof} \\
 a_4 : \text{Sunroof} = \text{sr2} &\xrightarrow{\text{incl}} \text{Opener} & a_8 : \text{Battery} = \text{small} \wedge \text{Engine} = \text{small} &\xrightarrow{\text{excl}} \text{AirCond}
 \end{aligned}$$

### Compatibility Constraints

- $$\begin{aligned}
 c_9 : \text{Package} = \text{standard} &\rightarrow \text{AirConditioner} \neq \text{ac2} \\
 c_{10} : \text{Package} = \text{luxury} &\rightarrow \text{AirConditioner} \neq \text{ac1} \\
 c_{11} : \text{Package} = \text{standard} &\rightarrow \text{Frame} \neq \text{convertible} \\
 c_{12} : (\text{Opener} = \text{auto}, \text{AirConditioner} = \text{ac1}) &\rightarrow \text{Battery} = \text{medium} \\
 c_{13} : (\text{Opener} = \text{auto}, \text{AirConditioner} = \text{ac2}) &\rightarrow \text{Battery} = \text{large} \\
 c_{14} : (\text{Sunroof} = \text{sr1}, \text{AirConditioner} = \text{ac2}) &\rightarrow \text{Glass} \neq \text{tinted}
 \end{aligned}$$

Figure 2-ii: Conditional CSP representation of the car configuration example

excluded from problem solutions. For example, activity constraint  $a_1$ :

$$a_1 : \text{Package} = \text{luxury} \xrightarrow{\text{incl}} \text{Sunroof}$$

involves variables *Package* and *Sunroof*, has the activation condition  $\text{Package} = \text{luxury}$ , and may include target variable *Sunroof* into the set of active variable if *Package* is successfully assigned or instantiated with value *luxury* and *Sunroof*'s activity status is undefined. The same activation condition is used in the activity constraint  $a_2$  to add *Sunroof* to the problem search space. Another form of activity control is exemplified by  $a_7$ :

$$a_7 : \text{Frame} = \text{convertible} \xrightarrow{\text{excl}} \text{Sunroof}.$$

If a configuration solution has *convertible* value for *Frame*, then that configuration excludes the *Sunroof* component altogether.  $a_8$  is another example of exclusion activity constraint. Finally, activity constraints can extend the set of active variable, that is, include or exclude variables, based solely on the activation status of some variables. One example is  $a_6$ :

$$a_6 : \text{Sunroof} \xrightarrow{\text{incl}} \text{Glass}.$$

The *Sunroof* variable, if active, regardless of the value it takes on, always adds *Glass* to the set of active variables, This means that if *Sunroof* participates in a configuration solution, *Glass* must participate too.

The compatibility constraints restrict the allowed value combinations for the variables on which the constraints are defined. One example is  $c_{11}$ :

$$c_{11} : \text{Package} = \text{standard} \rightarrow \text{Frame} \neq \text{convertible}$$

is a binary constraint defined on two variables: *Package* and *Frame*. It disallows the value *convertible* for *Frame* if *Package* takes on the value *standard*.

△

### 2.1.2 Definitions and Notations

The extension to the standard CSP paradigm we formalize in this section was originally called dynamic constraint satisfaction problem (DCSP), and was introduced by Mittal and Falkenhainer (Mittal & Falkenhainer 1990). They observed that configuration and model synthesis tasks render subsets rather than the entire set of problem variables relevant to final solutions. It means that not all variables need be assigned values in the course of problem solving. This type of dynamicity contrasts with another situation that bears the same name in the literature but refers to changing the set of variables independently of the solving process and, thus, assigning values to different sets of variables that correspond to changing the problem over time (Dechter & Dechter 1988), (Bessiere 1991), (Verfaillie & Schiex 1994). To distinguish between these two types of dynamic CSPs we

- rename Mittal and Falkenhainer's DCSP to *Conditional Constraint Satisfaction Problem* to
- capture the nature of the condition-based mechanism that controls change.

Central to conditional CSP is the notion of variable activity. Only *active variables* take on values and may make up solutions if value assignments are consistent over all problem constraints.

**Definition 1 (Active variable).** *A variable  $v$  is active iff  $v$  must be part of a solution<sup>1</sup>.*

To arrive at the definitions of consistent value assignments and problem solutions for conditional CSP class, we recall four basic definitions and some related notations that are common place in standard CSP class: definition of a standard CSP, variable instantiation (or value assignment), consistent instantiation, and solution to a CSP. For a comprehensive overview of the foundations of constraint satisfaction that consolidates the major results of CSP research and introduces terminologies, otherwise extremely diverse, used throughout constraint satisfaction community, we recommend the reader Tsang's book, *Foundations of Constraint Satisfaction* (Tsang 1993). The definitions below use Freuder's formalization (Freuder 1978).

**Definition 2 (Constraint satisfaction problem).** *A constraint satisfaction problem (CSP),  $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ , involves a finite set of variables,  $\mathcal{V} = \{v_1, \dots, v_n\}$ , which take on discrete values from their corresponding finite domains,  $\mathcal{D} = \{D_{v_1}, \dots, D_{v_n}\}$ , and a finite set of constraints,  $\mathcal{C} = \{c_1, \dots, c_m\}$ , which limit the value combinations that variables are allowed to take. Each constraint  $c_i \in \mathcal{C}$  is defined on a subset of  $k$  variables,  $\text{var}(c_i) = \{v_{i_1}, \dots, v_{i_k}\} \subseteq \mathcal{V}$ , and specifies allowed  $k$ -tuples or combinations of values that are a subset of the Cartesian product of the domains of constraint variables  $\text{var}(c_i)$ , that is,  $c_i \subseteq D_{v_{i_1}} \times \dots \times D_{v_{i_k}}$ .*

---

<sup>1</sup>Note that in a standard CSP all variables are active.

**Definition 3 (Variable instantiation or value assignment).** *An instantiation of a variable  $v$  is the assignment to  $v$  of a value  $d$  from the variable domain of values  $D_v$ , that is,  $v = d$ ,  $d \in D_v$ . We denote a variable instantiation or value assignment by the assignment  $v = d$  or the variable-value pair  $(v, d)$ . An instantiation of a set of variables  $V = \{v_1, \dots, v_k\}$  is the simultaneous instantiation of all variables in the set  $V$  with values from their associated domains, that is,*

$$\{v_1 = d_1, \dots, v_k = d_k\}, v_i \in V, d_i \in D_{v_i}, 1 \leq i \leq k.$$

*We denote an instantiation of a set of variables by the set of ordered pairs  $\{(v_1, d_1) \dots (v_k, d_k)\}$  or, more simply, by the  $k$ -tuple of assigned values  $(d_1, \dots, d_k)$ .*

**Definition 4 (Consistent instantiation or satisfied constraint).** *An instantiation  $\mathcal{I}_V$  of a set variables  $V$  is consistent with or satisfies a constraint  $c$  defined on the same set of variables,  $\text{var}(c) = V$ , if and only if  $\mathcal{I}_V \in c$ . An instantiation  $\mathcal{I}_V = (d_{v_1}, \dots, d_{v_n})$  satisfies a constraint  $c_U$  defined on a subset  $U \subseteq V$ ,  $|U| = m \leq |V| = n$ , if  $\mathcal{I}_U = (d_{u_1}, \dots, d_{u_m}) \in c_U$  and  $\{d_{u_1}, \dots, d_{u_m}\} \subseteq \{d_{v_1}, \dots, d_{v_n}\}$ . We call  $\mathcal{I}_U$  the instantiation  $\mathcal{I}_V$  restricted to  $U$ . An instantiation  $\mathcal{I}_V = (d_{v_1}, \dots, d_{v_n})$  satisfies a constraint  $c_W$  defined on a superset  $W \supseteq V$ ,  $|W| = p \geq |V| = n$ , if there is  $\mathcal{I}_W = (d_{w_1}, \dots, d_{w_p}) \in c_W$  and  $\{d_{w_1}, \dots, d_{w_p}\} \supseteq \{d_{v_1}, \dots, d_{v_n}\}$ . We call  $\mathcal{I}_V$  the instantiation  $\mathcal{I}_W$  extended to  $V$ .*

**Definition 5 (Solution to a CSP).** *A solution to a constraint satisfaction problem  $\mathcal{P}$  is a consistent instantiation of all variables in  $\mathcal{P}$ .*

A conditional CSP delimits from the entire set of problem variables a non-empty subset of *initial variables*. By definition these variables are active, *i.e.* they must participate in all solutions. A solving algorithm instantiates initial variables and checks the constraints that involve these variables. Some of the constraints are the traditional constraints in standard CSP and restrict variable instantiations. To differentiate them from the other type of constraints that model problem change in a conditional CSP, Mittal and Falkenhainer call these regular constraints *compatibility constraints*.

**Definition 6 (Consistent instantiation of a compatibility constraint).** A *compatibility constraint*,  $c$ , defined on the set of variables,  $\text{var}(c) = V_c$ , is consistent with an instantiation of these variables,  $\mathcal{I}$  of  $V_c$ , iff

- not all constraint variables are active, in which case we say that  $c$  is trivially satisfied by  $\mathcal{I}$ . Or,
- all constraint variables are active and  $\mathcal{I}$  is consistent with  $c$ .

Specific to conditional CSP are *activity constraints*. These constraints control the set of active variables, initialized with the set of initial variables. Activity constraints condition variables to either extend or restrict the set of active variables. *Inclusion activity constraints* condition variables to become active and be included or added to the set of active variables, and thus to the problem search space. *Exclusion activity constraints* condition variables to be not active and exclude them from being considered for activation by other activity constraints.

**Definition 7 (Inclusion activity constraint).** An *inclusion activity constraint*

$$a : a_{\text{cond}} \xrightarrow{\text{incl}} v_t$$

is composed of an activation condition,  $a_{\text{cond}}$ , which is a regular constraint defined on a set of condition variables  $V_{\text{cond}}$ , and target variable,  $v_t \notin V_{\text{cond}}$ , which is made active if and only if all condition variables are active and the instantiation of  $V_{\text{cond}}$  is consistent with  $a_{\text{cond}}$ .

**Definition 8 (Consistent instantiation of an inclusion activity constraint).** Given the inclusion activity constraint  $a : a_{\text{cond}} \xrightarrow{\text{incl}} v_t$ , which involves condition variables  $V_{\text{cond}}$  and target variable  $v_t$ , an instantiation  $\mathcal{I}$  of  $V_{\text{cond}}$  is consistent with  $a$  iff

- not all condition variables are active or  $\mathcal{I}$  is inconsistent with  $a_{\text{cond}}$ . We say that  $a$  is trivially satisfied by  $\mathcal{I}$ . Or
- all condition variables are active,  $\mathcal{I}$  satisfies  $a_{\text{cond}}$ , and  $v_t$  is active.

**Definition 9 (Exclusion activity constraint).** *An exclusion activity constraint*

$$a : a_{cond} \xrightarrow{\text{excl}} v_t$$

*is composed of an activation condition,  $a_{cond}$ , which is a regular constraint defined on a set of condition variables  $V_{cond}$ , and target variable,  $v_t \notin V_{cond}$ , which is made not active if and only if all condition variables are active and the instantiation of  $V_{cond}$  is consistent with  $a_{cond}$ .*

**Definition 10 (Consistent instantiation of an exclusion activity constraint).** *Given the exclusion activity constraint  $a : a_{cond} \xrightarrow{\text{excl}} v_t$ , which involves condition variables  $V_{cond}$  and target variable  $v_t$ , an instantiation  $\mathcal{I}$  of  $V_{cond}$  is consistent with  $a$  iff*

- *not all condition variables are active or  $\mathcal{I}$  is inconsistent with  $a_{cond}$ . We say that  $a$  is trivially satisfied by  $\mathcal{I}$ . Or*
- *all condition variables are active,  $\mathcal{I}$  satisfies  $a_{cond}$ , and  $v_t$  is not active.*

**Definition 11 (Activity constraint).** *An activity constraint is either an inclusion activity constraint or an exclusion activity constraint.*

Given an activity constraint  $a : a_{cond} \longrightarrow v_t$ , in the rest of the thesis we will use the following notations:

- $cond(a)$  denotes the set of condition variables,  $V_{cond}$
- $target(a)$  denotes  $v_t$ , the target variable associated with  $a$ .

**Definition 12 (Conditional constraint satisfaction problem [Mittal and Falkenhainer<sup>2</sup>]).** *A conditional constraint satisfaction problem,  $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_{\mathcal{I}}, \mathcal{C}_{\mathcal{C}}, \mathcal{C}_{\mathcal{A}} \rangle$ , involves a finite set of variables,  $\mathcal{V} = \{v_1, \dots, v_n\}$ , which, if active, can take on discrete values from their corresponding finite domains  $\mathcal{D} = \{D_{v_1}, \dots, D_{v_n}\}$ , a non-empty set of initially active*

---

<sup>2</sup>Name changed from original dynamic CSP to avoid confusion with another dynamic CSP class where dynamic changes occur independently of problem formulation.

variables, called *initial variables*,  $\mathcal{V}_I$ ,  $\mathcal{V}_I \subseteq \mathcal{V}$ , a set of *compatibility constraints*  $\mathcal{C}_C$ , and a set of *activity constraints*  $\mathcal{C}_A$ .

**Definition 13 (Solution to a conditional CSP).** A solution,  $sol$ , to a conditional constraint satisfaction problem,  $\mathcal{P}$ , is an instantiation of a set of active variables,  $V_{sol}$ , such that  $sol$  is consistent with all compatibility constraints that involve  $V_{sol}$ , and is consistent with all activity constraints whose activation conditions involve  $V_{sol}$ .

We observe that, in the course of solving a conditional CSP with  $V$  variables, some instantiation  $\mathcal{I}$  of active variables partitions  $V$  according to the activity constraints  $\mathcal{I}$  satisfies. Activity constraints of inclusion consistent with  $\mathcal{I}$  construct the set of *included variables*,  $V_{incl}$ . Included variables are active and must participate in problem solutions. Activity constraints of exclusion consistent with  $\mathcal{I}$  construct the set of *excluded variables*,  $V_{excl}$ ,  $V_{excl} \cap V_{incl} = \emptyset$ . Excluded variables are not active and cannot be made active by subsequent inclusion activity constraints that are consistent with  $\mathcal{I}$ . This means that  $\mathcal{I}$  cannot be extended with variables in  $V_{excl}$  to form complete solutions. Similarly, included variables are active and cannot be excluded by subsequent exclusion activity constraints that are consistent with  $\mathcal{I}$ . The remaining variables  $V_{rem} = V - (V_{incl} \cup V_{excl})$  are not active and their participation in problem solutions that extend  $\mathcal{I}$  has not been determined.

We represent this information by associating an *activity status* property with each variable. Accordingly, a variable can be in one of the following states:

1. *initial*, if the variable is part of the initial variable set,
2. *included*, if the variable has been added to the set of active variables by an inclusion activity constraint,
3. *excluded*, if the variable has been excluded from the set of active variables by an exclusion activity constraint, and
4. *undefined*, if the variable does not fit in any of the above three categories.

**Definition 14 (Activity graph).** Given a conditional CSP  $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ , an *activation graph* is a directed graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  where the vertex set is the set of variables  $\mathcal{V}$

and the edge set,  $\mathcal{E}$ , is the set of edges  $(v_i, v_t)$  with  $v_i, v_t \in \mathcal{V}$  such that there is an activity constraint  $a \in \mathcal{C}_A$  with  $v_i \in \text{var}(a)$  and  $v_t = \text{target}(a)$ . An **inclusion activation graph** is an activity graph which considers only the inclusion activity constraints.

**Definition 15 (Path).** A path of length  $k$  from a vertex  $u$  to a vertex  $u'$  in the activity graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  for a conditional CSP  $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$  is a sequence  $\langle v_0, v_1, v_2, \dots, v_k \rangle$  of vertices such that  $u = v_0$ ,  $u' = v_k$ , and  $(v_{i-1}, v_i) \in \mathcal{E}$ , for all  $i = 1, 2, \dots, k$ . The **length** of the path is the number of edges in the path. The path **contains** the vertices  $v_0, v_1, v_2, \dots, v_k$  and the edges  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ . If there is a path  $p$  from  $u$  to  $u'$  we say that  $u'$  is **reachable** from  $u$  via  $p$ . A path  $\langle v_0, v_1, v_2, \dots, v_k \rangle$  forms a cycle if  $v_0 = v_k$  and the path contains at least one edge. The cycle is **simple** if, in addition,  $v_0, v_1, v_2, \dots, v_k$  are distinct. An activity graph with no cycles is **acyclic**.

**Definition 16 (Activity cycle).** An activity cycle in a conditional CSP  $\mathcal{P}$  is a cycle in  $\mathcal{P}$ 's inclusion activity graph.

**Definition 17 (Activation path).** Given a conditional CSP problem  $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ , an **activation path** of length  $k$  is a path in the activity graph such that:

- $v_0 \in \mathcal{V}_I$ .
- $p$  forms no cycles, and
- $p$  is maximal, i.e.  $p$  cannot be extended to an activation path  $p'$  of length  $k' > k$ .

## 2.2 Random Problem Generation

### 2.2.1 Introduction

The simplicity of the constraint satisfaction model has led to relevant quantitative characterizations of CSP instances that can be used as parameters for automatic generation of random CSPs. In the case of standard binary CSPs, relevant measures of problem properties are the number of variables, number of constraints, domain size, and number of value pairs

included in a constraint. By systematically varying these parameters using some random methods, large sets of standard binary CSPs can be generated for the purpose of conducting useful experimental analyzes.

In this thesis we present extensive empirical evaluations of the proposed solving methods. The reformulation algorithm is also evaluated experimentally. The evaluations make use of randomly generated conditional CSPs. This section describes the model of problem generation for binary standard CSPs and its adaptation to generate the specialized class of random conditional CSPs.

### 2.2.2 Random Standard CSPs

In our experiments we used Freuder and Wallace's model of constant probability of inclusion for generating random CSPs (Freuder & Wallace 1992). In this model, the *number of variables*,  $n$ , and *maximum domain size*,  $d_{max}$ , are fixed. These two parameters allow for delimiting the range within which we can vary the actual number of elements included into variable domain, constraint set, and allowed value pairs set of a constraint. Thus, for a binary CSP problem instance, the *actual number of domain values* can vary between 1 and  $d_{max}$ . The *actual number of constraints* can range from  $n - 1$  to  $(n * (n - 1))/2$  constraints. There is a minimum number of  $n - 1$  constraints in a constraint graph that is reduced to a line connecting all variable nodes. The maximum number of constraints in a complete constraint graph is  $(n * (n - 1))/2$ . Finally, the *actual number of allowed pairs in a binary constraint* can vary between none and all possible value combinations between the two domains on which the constraint is defined, that is, from 0 to the cardinality of the Cartesian product of the constraint variable domains. However, the generator has initialization options for removing empty and full-product constraints.

The actual number of domain values, constraints, and allowed value pairs are varied using probabilities. Thus, there are three probabilities for generating these problem elements. The probability of constraint inclusion is also known as characterizing the *density* of the problem, while the probability of including value pairs in a constraint determines the

problem *satisfiability*.

We are presenting the same example given in (Freuder & Wallace 1992) to explain how the method works for the choice of number of constraints. If the set of problems is characterized by  $n = 10$  variables, which already are connected such that the constraint graph is a line (there should be at least 9 binary constraints), the maximum number of constraints that can be added to this initial problem instance is  $(10 \cdot 9)/2 - 9 = 36$ . Assume that the probability of inclusion of a single constraint is fixed at 0.3. The set of problems of size 10 has an expected value for the number of constraints of  $(36 \cdot 0.3) + 9 \approx 20$ . The generator builds at random a constraint graph that has 20 constraints. A spanning tree phase is added to insure that the generated graph is connected. The constant probability of inclusion model uses the same method to determine the domain size and the number of value pairs included in a constraint.

For full instructions on the use of the random CSP generator and a description of other underlying models, we direct the user to the technical manual posted on the web site of the Constraint Computation Center at University of New Hampshire, with which Wallace was affiliated at the time he developed these tools (Wallace 1996).

### 2.2.3 Random Conditional CSPs

Random conditional CSP inherits from random standard CSP all five parameters described in the model of constant probability of inclusion: number of variables, maximum domain size, probability of domain value inclusion, probability of constraint inclusion, or *density*  $d$ , and probability of constraint value pair inclusion, or *satisfiability*  $s$ . New parameters are needed, however, to characterize problem activity and to measure:

- the amount of activity condition constraints induce,
- how “active” a variable domain is,
- the type of activity that takes place, i.e., whether variables are included or excluded from a problem instance,

- activation redundancy, i.e., the number of activations that target the same variable and whether these activations involve different sets of condition variables,
- the size of the initial variable set,
- whether activated variables, in turn, trigger more activation.

To control these parameters, Richard Wallace has extended the model of constant probability of inclusion for standard binary CSPs with nine additional parameters that collect activity information for a specialized class of conditional binary CSP. The class restricts both compatibility and activity constraints constraints to binary constraints. Binary activity constraints are defined on a single condition variable and the usual target variable. Since condition constraints are reduced to single value assignments, we call these values *condition values*.

The activity parameters, similarly to standard parameters, set maximum limits on sizes of problem component sets, indicate probabilities of inclusion of certain activity elements, and answer true/false questions about combining the effect of activity parameters. The definitions and notational abbreviations of these parameters are:

- Maximum number of condition values per domain, *maxCondPerDom*, sets the maximum number of conditional values per domain;
- Total number of condition values per problem instance, *totalCond*, may set a stricter limit than the total derived from multiplying *maxCondPerDom* and the number of domains. As soon as the imposed *totalCond* is reached, the problem generator leaves the remaining non-initial variables with no condition values. Thus, condition values are not uniformly distributed throughout the entire problem;
- Maximum number of target variables per condition value, *maxTargetPerCond*, sets the maximum number of target variables one condition value can include or exclude from a problem;

- Probability of generating a non-initial variable as a target variable. The number of target variables is a basic indicator of the density of the activity constraint graph, although it does not produce the actual number of activity constraints. Even if it is not the direct counterpart of the *density* of the compatibility constraint graph, we call it *density of activity* and denote it by  $d_a$ . We rename the standard density  $d_c$  to signal its relationship with compatibility constraints.
- Probability of generating a value in a domain as a condition value. The number of condition values measure the satisfiability of the activation condition, a unary constraint defined on that domain. We call this probability *satisfiability of activation* and denote it by  $s_a$ . We rename its standard counterpart  $s_c$  since it refers more specifically to the satisfiability of the compatibility constraints.
- Probability of generating an activity constraint as activity constraint of inclusion,  $p_a$ . The probability of generating an activity constraint of exclusion is given by  $1 - p_a$ .
- No condition value in the domain of target variables, *noCondInTarget*, is enforced when true; otherwise, active variables trigger more activations via their condition values;
- No activation redundancy produced by condition values in different domains, *noRd-ntDiffDom*, is enforced when true; otherwise, condition values assigned in different condition variables can target the same variable for activity status change;
- No activation redundancy produced by condition values in the same domain, *noRd-ntSameDom*, is enforced when true; otherwise, condition values in the same domain can target the same variable.

In all experimental analyses throughout this thesis we use the random problem generator for conditional binary CSPs written by Richard Wallace. The generator collects two sets of parameters. The first set contains five standard parameters used to generate random standard CSPs and the number of problem instances with these characteristics. The second

set has nine dynamic parameters based on which the preliminary, underlying standard CSP is transformed into a conditional binary CSP.

As a general practice, the most prevalent experimental design for studying algorithm performance using random standard CSPs involves varying density and satisfiability. For conditional CSPs we refer to these parameters as *density of compatibility*,  $d_c$ , and *satisfiability of compatibility*,  $s_c$ .

Specific to a conditional CSP, we are interested in generating combinations of parameter values for those dynamic parameters that control the amount of activity produced. The most salient parameters of problem activity are *density of activity*,  $d_a$ , and *satisfiability of activation*,  $s_a$ .

Moreover, there is no restriction on problem activity as controlled by the three boolean parameters:

- *noCondInTarget* is set to false to allow active variables to have activating values,
- *noRdntDiffDom* and *noRdntSameDom* are set to false to permit redundant activity constraints: activity status of the same variable can be determined by condition values assigned in domains of either different condition variables or the same condition variables.

With the probability of generating inclusion vs. exclusion activity constraints,  $p_a$ , we control the problem activity's expansionist character, when more variables are made active, versus the problem activity's conservative character, when more variables are restricted from being active. This parameter also measures the intrinsic tension between the two opposing types of activity constraints. If  $p_a$  is set to 0.5, it is more likely that the problem has conflicting activations. This situation occurs when the same variable is both included into and excluded from the problem. As the number of inclusion and exclusion activity constraints is the same, it is more likely that such conflicts occur.

There are three more parameters required by conditional CSP generation: *maxCondPerDom*, *totalCond*, and *maxTargetPerCond* set maximum limits for condition values per domain

and whole problem, and maximum target variables per condition value. They are used to fix the range of variability for the probability parameters.

## CHAPTER 3

### SOLVING METHODS

#### 3.1 Introduction

The standard constraint satisfaction domain benefits from a rich collection of field-tested solving methods. In contrast, solving conditional CSP is still in its infancy with very little research directed to specialized solving methods. Following the model of other CSP specializations, we developed adaptations of the most representative standard CSP methods for the conditional domain:

1. a modified backtracking (BT) search algorithm that handles both types of activity constraints,
2. a new forward checking (FC) algorithm that propagates compatibility constraints over active variables,
3. a new maintaining arc-consistency (MAC) algorithm that propagates both compatibility and activity constraints.

In the next chapter, the relative performance of the proposed methods is analyzed experimentally by using random conditional CSPs. We show that the run-time complexity order in the standard domain,  $BT < FC < MAC$ , holds in the conditional domain. The advantage that maintaining arc-consistency has over forward checking is due in part to the propagation of the activity constraints.

Backtrack search is the only algorithm that has been previously adapted for conditional constraint satisfaction (Gelle 1998). Its implementation handles directly only activity constraints of inclusion. Activity constraints of exclusion are reformulated as compatibility

constraints based on Haselböck’s transformation introduced in (Haselböck 1993). We modify the algorithm to handle both types of activity constraints as given in the original problem representation. The forward checking and maintaining arc consistency algorithms for conditional CSPs that are presented in this chapter are new. Reference to “extending a conventional backtrack search CSP” with “forward checking to propagate all constraints” has been made in (Mittal & Davis 1989). However, no description is given of either the backtrack search or forward checking algorithm, and no explanation is provided as to what this propagation means for each type of constraints.

The chapter’s organization has three sections that present the proposed solving methods, followed by a chapter summary section. For each method we first show an execution trace of the algorithm on a sample problem. We then proceed with the description of the algorithm and its procedures in pseudocode. The sample problem we use in this chapter (Example 3) is a subproblem of the CSP model of the car configuration in Example 1 in the previous chapter.

### Example 3.

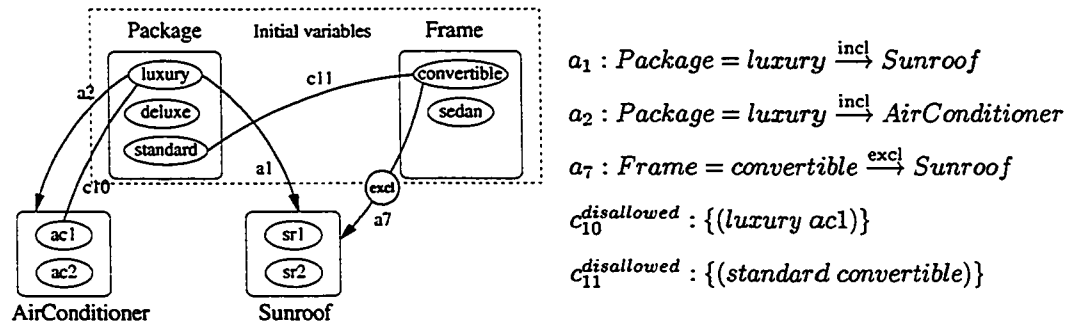


Figure 3-i: Conditional CSP subproblem from the car configuration in Example 1

The example has four variables: *Package*, *Frame*, *Sunroof*, and *AirConditioner*, with their associated domains of values. *Package* and *Frame* initialize the search space, while *Sunroof* and *AirConditioner* might be included or excluded from the search space. Variable *AirConditioner* is conditionally included in the search space through the activity

constraint  $a_2$ . Variable *Sunroof*'s inclusion is conditioned by  $a_1$  activity constraint, while its exclusion is conditioned by  $a_7$  activity constraint. The two compatibility constraints further restrict the combinations of allowed values.

△

## 3.2 Backtrack Search

Backtrack search for conditional constraint satisfaction is a natural extension of backtrack search for standard constraint satisfaction. After it was first mentioned in (Mittal & Falkenhainer 1990), a full description of a backtrack solving algorithm for conditional CSPs was presented in (Gelle 1998). The algorithm, however, solves a partially reformulated conditional CSP, which has only activity constraints of inclusion. We present a modified version of the algorithm that handles both types of activity constraints. In comparison with its standard analogue, backtrack search in the context of conditioning portions of the search space through activity constraints adds to compatibility checking the consistency checking of the activity constraints.

### 3.2.1 Example

We examine first an example of how the backtrack search method works. Figure 3-ii shows the depth-first search traversal of the search tree for finding all solutions to the sample problem in Example 3.

**Example 4.** The search starts with the initial variable *Frame*, for which it tries value *convertible*. This assignment is relevant only to the activity constraint  $a_7$ , whose condition involves *Frame* and targets variable *Sunroof*. All the other constraints are trivially satisfied or irrelevant since they involve variables which are not instantiated yet. Thus, there is only one constraint check, for  $a_7$ . Its condition is satisfied, that is, *Frame* is allowed to take *convertible*, and causes the exclusion of *Sunroof* variable from the search subtree rooted at instantiation *Frame* = *convertible*. We move on to the other initial variable and

try *luxury* for *Package*. Again, no compatibility constraint is relevant to this instantiation. The assignment is relevant to  $a_1$ , which includes *Sunroof*, but this variable has already been excluded by assigning *convertible* to *Frame* and satisfying  $a_7$ . Thus,  $a_1$  fails, renders *luxury* inconsistent, so we have to try the next value, *deluxe*, for *Package*. This one holds, the search space is not extended further, and we have the first solution to the problem: (*convertible*, *deluxe*).

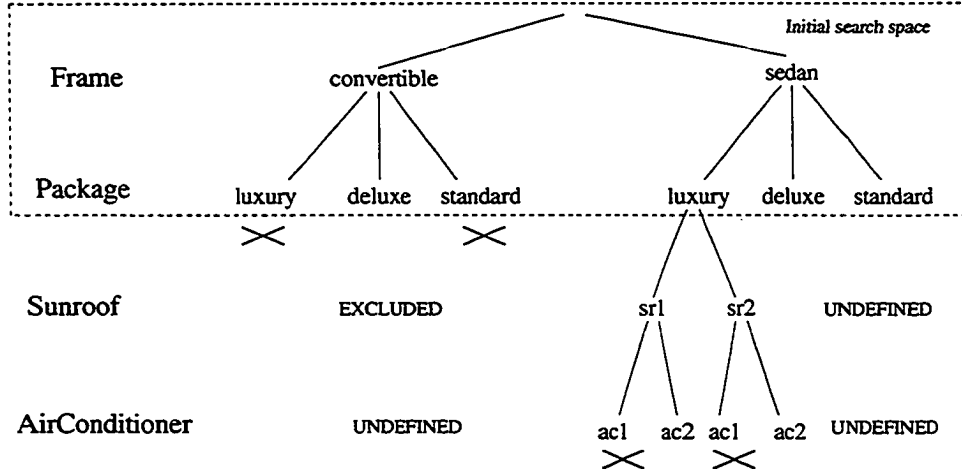


Figure 3-ii: Backtrack search trace on the sample problem in Example 3

To find all solutions, we continue and try value *standard* for *Package*. The assignment violates the compatibility constraint  $c_{11}$ , so we back up one level to the top of the search tree, we reset the activity status of *Sunroof* to undefined, and instantiate *Frame* with *sedan*. There is no relevant compatibility constraint or activity constraint for this assignment, so we continue with trying *luxury* for *Package*. Again activity constraints  $a_1$  and  $a_2$  are checked, both hold, so *Sunroof* and *AirConditioner* are activated. We try *sr1* for *Sunroof*, no constraint needs to be checked, so we proceed with trying *ac1* for *AirConditioner*. The assignment violates  $c_{10}$ , and we have to try the other value, *ac2*, next. No constraint restricts this assignment and no activity is defined for this value. Thus we obtain the solution, (*sedan*, *luxury*, *sr1*, *ac2*). Similarly, by assigning *sr2* to *Sunroof*, we obtain the solution

$(\text{sedan}, \text{luxury}, \text{sr}_2, \text{ac}_2)$ . We back up two levels, deactivate *Sunroof* and *AirConditioner*, try the other values for *Package*, which are not constrained by any compatibility or activity constraint, and determine the last two solutions:  $(\text{sedan}, \text{deluxe})$  and  $(\text{sedan}, \text{standard})$ .

△

### 3.2.2 Algorithm

Checking constraint consistency in a conditional context entails checking consistency for both types of constraints: compatibility and activity constraints. When a new *value* is tried for a variable *var* during search, the backtrack search algorithm determines whether the constraints involving *var* are satisfied. The algorithm we propose delegates consistency checking at the time of instantiating *var* to two different procedures, corresponding to the two types of constraints.

#### Checking compatibility constraints

A compatibility constraint that involves the currently instantiated variable, *var*, is checked only if all the other variables on which that constraint is defined are already instantiated. We call the problem's instantiated variables *past variables*. Otherwise, no compatibility constraint check is performed as the constraint is considered trivially satisfied.

Algorithm 3.1, *BtCompatibility*, implements the compatibility constraint check performed when *value* is tried for some active variable *var*. The compatibility constraint check holds for a given constraint if *value* is consistent with the value assignments of the past variables that the constraint involves. Without restricting this check to binary constraints, *value* assignment is consistent over a compatibility constraint if *value* participates in at least one allowed value tuple of that constraint.

#### Checking activity constraints

An activity constraint that involves the currently instantiated variable *var* is checked for consistency only if the constraint condition is defined on *var* and past variables, and the

**Algorithm 3.1.** *Consistency check for compatibility constraints used in backtrack search.*

```

boolean BtCompatibility(var, value) {
   $C \leftarrow$  compatibility constraints which involve variable var
  for each ( $c \in C$ ) {
    let  $V_c$  be  $c$ 's variables other than var
    if ( $V_c$  are not all past variables)
      return true //  $c$  is trivially satisfied; no check is done
    if (value is consistent with value assignments of  $V_c$ )
      return true
    return false // var's instantiation with value fails
  } // end for
} // end BtCompatibility()

```

condition holds. Otherwise, the activity constraint is disregarded as trivially satisfied. If *value* is consistent with the value assignments of the other condition variables, then the activity constraint may change the search space by either including or excluding target variables. The search path leading to *var* has only past variables. The search tree rooted at *var* has only included variables that are not instantiated yet. We call the problem's included but not instantiated variables *future variables*.

To keep track of how the search space changes when activity constraints of inclusion are found consistent, we maintain a list of future variables, *Agenda*. The list is initialized with the problem's initial variables, the only future variables that the problem has initially. During search, one variable at a time is removed from the *Agenda* and instantiated.

If the type of *action* triggered when an activity constraint is satisfied is contradicted by the activity status of the *target* variable, then the activity constraint check fails. That is the case when the constraint action either includes a target variable, but that target variable is already excluded, or excludes a target variable, but that target variable is already included in the search space.

Algorithm 3.2, *BtActivity*, describes the checking of activity constraints. The implementation has two cases, controlled by the type of *action* that activity constraints perform. The two possible actions are to either include or exclude a *target* variable. In each case, if *action* matches the activity status of the *target* variable, the activity constraint is redun-

**Algorithm 3.2.** *Consistency check for activity constraints used in backtrack search.*

```

boolean BtActivity(var, value, Agenda, UndoActivity) {
  A  $\leftarrow$  activity constraints whose conditions involve variable var
  for each (a  $\in$  A) {
    let Vcond be a's condition variables except var
    if (Vcond are not all past variables)
      return true // a is trivially satisfied; no check is done; successful termination
    if (value is not consistent with value assignments of Vcond)
      return true // activation condition fails; no effect on variables' activity status
      //successful termination
    else {
      target  $\leftarrow$  target variable of a
      action  $\leftarrow$  activity performed by a
      if ( action includes target ) {
        if ( target has already been excluded )
          return false //conflicting activity constraints
        else if ( target is newly included )
          Agenda  $\leftarrow$  Agenda  $\cup$  target //target becomes future variable
        }
      else { // action excludes target
        if ( target has already been included )
          return false //conflicting activity constraints
        }
      UndoActivity  $\leftarrow$  UndoActivity  $\cup$  {a}
    } //end else
  } //end for
  return true
} //end BtActivity()

```

dant and has no effect on the search space. Otherwise, the activity constraint generates a conflict and fails. The third possibility is when *target*'s status is undefined. As a result of enforcing the activity constraint, the *target* is newly included or excluded. If *target* is included, it is added to the search space. *BtActivity* procedure adds *target* to the *Agenda* and changes the variable status from undefined to included. If *target* is excluded from the search space, *Agenda* remains unchanged, but the variable status changes to excluded. In both cases, *UndoActivity* remembers the activity constraint that modified the search space. The search space is restored based on this information when variable instantiation fails.

**Algorithm 3.3.** *Backtrack algorithm for solving conditional CSPs.*

```

boolean CCSP_SolveBt( ) {
    Agenda  $\leftarrow$  initially included variables
    if ( Agenda is empty )
        return false
    numberSolutions  $\leftarrow$  0
    oneSolutionOnly  $\leftarrow$  whether one or all solutions are searched for
    return CCSP_Bt(Agenda, oneSolutionOnly)
} //end CCSP_SolveBt

boolean CCSP_Bt(Agenda, oneSolutionOnly) {
    if ( Agenda is empty ) {
        numberSolutions  $\leftarrow$  numberSolutions + 1
        return true
    }
    var  $\leftarrow$  select variable and remove from Agenda
    value  $\leftarrow$  select value from domain of var and instantiate var
    UndoActivity  $\leftarrow$   $\emptyset$ 
    UndoValues  $\leftarrow$   $\emptyset$ 
    if ( BtCompatibility(var, value) and BtActivity(var, value, Agenda, UndoActivity) ) {
        if ( CCSP_Bt(Agenda, oneSolutionOnly) and oneSolutionOnly ) {
            reset variable activity status as saved in UndoActivity
            unstantiate var and put it back into the Agenda
            return true
        }
    }
    reset variable activity status as saved in UndoActivity
    unstantiate var and put it back into the Agenda
    UndoValues  $\leftarrow$  {(var, value)}
    remove value from domain of var
    if ( domain of var is empty )
        backtrackSearch  $\leftarrow$  false
    else
        backtrackSearch  $\leftarrow$  CCSP_Bt(Agenda, oneSolutionOnly)
    reset variable activity status as saved in UndoActivity
    restore all removed values saved in UndoValues
    return backtrackSearch
} //end CCSP_Bt()

```

***CCSP\_Bt* algorithm**

The Algorithm 3.3 shows the implementation of the procedure *CCSP\_SolveBt* for solving conditional CSPs using backtrack search. After initializing the *Agenda* with all initially included variables, and determining whether one or all solutions are wanted, the algorithm calls the recursive procedure *CCSP\_Bt*, which implements backtrack search. *CCSP\_Bt* traverses the search tree in a depth-first search fashion: going deeper in the tree by recursing

through the variables in the *Agenda* if both compatibility and activity constraints hold for the current instantiation. Recursion is also used to do a sideways traversal of the search tree when different values are, in turn, assigned to a given variable. More than one value is tried for a variable if previous instantiations failed. It is also the case that all domain values are checked when the algorithm seeks all solutions.

*BtCompatibility* and *BtActivity* procedures are called with each variable instantiation. If compatibility and activity constraint checks hold, the assignment of the current variable is kept and the search goes deeper in the tree by recursing through the next future variable in the *Agenda*. If a compatibility or activity constraint relevant to the current instantiation fails, the search goes sideways in the tree by recursing through the next value in the domain of the variable selected from the *Agenda* with the second recursive call. Prior to making that recursive call, the inconsistent value is removed from its domain and removal information is saved in *UndoValues*.

When recursion unwinds (either *Agenda* is empty or there is no value left in a domain), the traversal backs up in the tree, *UndoValues* is used to restore domains, and activity status changes are undone by processing *UndoActivity*. These bookkeeping operations are necessary especially when the algorithm searches for all solutions. Note that in the case of finding only one solution, when a solution is found the two bookkeeping statements that precede the successful return statement have no effect on the search result. They merely leave the problem as it was before search started.

*BtActivity* maintains undo information, *UndoActivity*, with regard to variables' activity statuses. This information has to be restored if the procedure fails. The order in which constraint checking is done has *BtCompatibility* before *BtActivity*, in order to avoid modifying activity status and the content of the *Agenda* unnecessarily if a compatibility constraint fails. We will see that for the solving algorithms that we present next checking compatibility constraints still requires less bookkeeping than checking activity constraints.

### 3.3 Forward Checking

Standard forward checking combines backtracking with some form of local consistency, or “look ahead”, that prunes values from variables which have not been assigned values in the search tree. (Haralick & Elliott 1980). The version of forward checking in conditional context enforces look-ahead consistency along compatibility constraints. Checking activity constraints might add new variables to the set of variables that await to be assigned values. Local consistency is propagated to these variables as well.

#### 3.3.1 Example

We use the same sample problem as in the previous section to show an execution trace of the forward checking method applied to conditional CSPs (Figure 3-iii).

**Example 5.** We observe that when *Frame* is the only instantiated variable and has been assigned the value *convertible*, forward checking examines all the other active, but not instantiated variables which share compatibility constraints with *Frame*. Variable *Package* is such a variable since it participates in the constraint  $c_{11}$  along with *Frame*, and it is the only variable in the search tree that is not currently instantiated. As a result, forward checking removes from the domain (*luxury*, *deluxe*, *standard*) of *Package* the value *standard*, which is inconsistent with the assignment *convertible* for *Frame*.

Analogous to backtrack search for conditional CSPs, the presence of activity control in addition to compatibility control, extends forward checking to the activity constraints. The only applicable activity constraint when *Frame* takes *convertible* is  $a_7$ , which excludes *Sunroof* from the search tree. Since *Sunroof* is not added to the agenda of search variables, no filtering of its domain takes place at this point. The algorithm continues in a depth-first search manner and instantiates variable *Package* with value *luxury*. There is no other variable in the search tree to look ahead to, so we continue with checking the activity constraints. The applicable activity constraints to the current instantiation of *Package* are  $a_1$  and  $a_2$ .  $a_1$  attempts to include *Sunroof*, conflicts with  $a_7$ , and thus fails. This means that *luxury* is not a valid choice,  $a_2$  is not checked anymore, and the algorithms



more solutions, unrestricted by any constraint: (*sedan, deluxe*), and (*sedan, standard*).

△

### 3.3.2 Algorithm

#### Forward checking compatibility constraints

Forward checking of compatibility constraints is done when a new *value* is tried for a variable *var*. Compatibility constraints are trivially forward-checked if they are defined on past variables and thus have already been instantiated. The actual forward checking takes place when variables other than *var* are future variables and thus have not yet been instantiated. The procedure *FcCompatibility* implements forward checking along compatibility constraints (Algorithm 3.4). If *value* for *var* is inconsistent with values in the domains of future variables, the inconsistent values are removed from their domains. Forward checking fails if one of those domains becomes empty. Otherwise, information about the removed values and their variables is saved in *UndoValues*. The undo information is needed to restore the domains affected by forward checking when search backs up and undoes the work it had performed up to that point in the search.

**Algorithm 3.4.** *Forward checking for compatibility constraints.*

```

boolean FcCompatibility(var, value, UndoValues) {
  C ← compatibility constraints which involve variable var
  for each (c ∈ C) {
    let Vc be c's variables other than var
    if (Vc are not all future variable)
      return true // c is trivially satisfied; no check is done
    for each (v ∈ Vc) { // v is a future variable
      let d be the domain of v
      if ( d has inconsistent values Id with value ) and (d − Id is empty) )
        return false
      remove Id from d and save (v, Id) information in UndoValues
    } // end for each v
  } // end for each c
  return true
} // end FcCompatibility()

```

Note that, similar to *BtCompatibility*, *FcCompatibility* procedure is not restricted to checking binary constraints. The propagation of the value assignment  $var = value$  to future variables along compatibility constraints that involve  $var$  results in the removal of those values in the future variable domains that are not part of the tuples that have  $value$  for  $var$ . We assume that compatibility constraints are represented as enumerations of allowed tuples.

### Forward checking activity constraints

Forward checking of activity constraints performs activity constraint checks under the same circumstances as backtrack search:

- compatibility constraints have been forward checked, and
- activity constraint conditions are defined on past variables and current variable,  $var$ , and hold for current instantiation,  $var = value$ .

Forward checking of activity constraints differs from its backtrack search counterpart when activity constraints extend the search tree with new active variables. In the presence of newly included variables, forward checking consistency filters from the new domains those values which are inconsistent with the current partial solution. If activity constraints exclude new variables, or introduce redundant activity, forward checking does not have any effect on the future variable set.

To implement forward checking for activity constraints, we slightly modify *BtActivity* we used in backtrack search such that it collects the newly added variables in a list *NewVariables*. If the current instantiation,  $var = value$ , does not introduce conflicts with regard to these variables' activity statuses, then the domains of the variables in *NewVariables* are possibly pruned of values inconsistent with the current search path.

The implementation of forward checking for activity constraints is shown in Algorithm 3.5. The algorithmic additions to *BtActivity* used in backtrack search are shown in boxes in the implementation of *FcActivity*. The procedure *FcActivity* defines a local

**Algorithm 3.5.** *Forward checking for activity constraints. The code enclosed in boxes shows how  $FcActivity$  implementation differs from  $BtActivity$  in Algorithm 3.2.*

```

boolean  $FcActivity(var, value, Agenda, UndoValues, UndoActivity)$  {
   $A \leftarrow$  activity constraints whose conditions involve variable  $var$ 
  for each ( $a \in A$ ) {
    let  $V_{cond}$  be  $a$ 's condition variables except  $var$ 
    if ( $V_{cond}$  are not all past variables)
      return true //  $a$  is trivially satisfied; no check is done
    if ( $value$  is not consistent with value assignments of  $V_{cond}$ )
      return true // activation condition fails; no effect on variables' activity status
    else {
       $target \leftarrow$  target variable of  $a$ 
       $action \leftarrow$  activity performed by  $a$ 
      if ( $action$  includes  $target$ ) {
        if ( $target$  has already been excluded)
          return false // conflicting activity constraints
        else if ( $target$  is newly included) {
           $Agenda \leftarrow Agenda \cup target$  // target becomes future variable
           $NewVariables \leftarrow NewVariables \cup target$ 
        }
      }
      else { //  $action$  excludes  $target$ 
        if ( $target$  has already been included)
          return false // conflicting activity constraints
      }
       $UndoActivity \leftarrow UndoActivity \cup \{a\}$ 
    } // end else
  } // end for

  
    for each ( $newvar \in NewVariables$ )
      if ( $FcNewvar(newvar, UndoValues)$  is false)
        return false
  

  return true
} // end  $FcActivity()$ 

```

list,  $NewVariables$ , which saves the newly included variables. If all activity constraints processed in the first loop are satisfied, then  $FcNewvar$  procedure (Algorithm 3.6) is called to propagate value assignments of past variables (including the current instantiation) to variables in  $NewVariables$ . The propagation takes place over compatibility constraints that connect the current search path with  $newvar$ . This is necessary because forward checking of activity constraints ( $FcActivity$ ) may modify the future variable set that has been forward-checked along compatibility constraints with  $FcCompatibility$ , prior to executing  $FcActivity$ .  $FcNewvar$  is a restricted form of  $FcCompatibility$  which deals with

*NewVariables* only and their consistency with the current search path.

**Algorithm 3.6.** *Filter the domain of newvar along compatibility constraints for which newvar is a future variable.*

```

boolean FcNewvar(newvar, UndoValues) {
   $C \leftarrow$  compatibility constraints that involve newvar
    and connect newvar to past variables (including current instantiation)
  for each ( $c \in C$ ) {
    let  $V_c$  be  $c$ 's variables except newvar
    let  $A_c$  be the value assignments of variables  $V_c$ 
    let  $d$  be the domain of newvar
    if (  $d$  has inconsistent values  $I_d$  with  $A_c$  ) and ( $d - I_d$  is empty) )
      return false
    else
      remove  $I_d$  from  $d$  and save (newvar,  $I_d$ ) in UndoValues
    }//end for
  return true
}//end FcNewvar()

```

An alternative to the use of *FcNewvar* is to apply *FcActivity* first and generate the set of future variables on which we then run *FcCompatibility*. The drawback of this approach is that in case *FcCompatibility* fails, not only do remove values have to be put back, but also the future variable space has to be restored: the content of the *Agenda* and variables' activity status.

### *CCSP\_Fc* algorithm

The recursive forward checking algorithm for solving conditional CSPs, *CCSP\_Fc* (Algorithm 3.7) is very similar to *CCSP\_Bt* backtrack algorithm. The main difference is in the way compatibility and activity constraints are used to check the consistency of the current instantiation. Instead of looking back along compatibility constraints to values already assigned and checking if the current value assignment satisfies these constraints, *FcCompatibility* looks ahead to variables not yet instantiated. It checks that domain values of future variables satisfy compatibility constraints these future variables share with the currently assigned variable. If *FcCompatibility* is successful, the algorithm checks the activ-

ity constraints by running *FcActivity* and filters the domains of the newly included variables such that they are consistent with the search path. In addition to calling the new constraint checking algorithms, *FcCompatibility* and *FcActivity* (shown in boxes in the *CCSP\_Fc* algorithm (Algorithm 3.7), the algorithm differs from the backtrack search when it restores removed values saved in *UndoValues*. That is necessary because both *FcCompatibility* and *FcActivity* prune future variables and save undo information in *UndoValues* list.

We notice that both types of constraint checks, *FcCompatibility* and *FcActivity*, might prune future domains and hence have to maintain information about removed values in *UndoValues*. As a result, when a current instantiation  $var = value$  violates either a compatibility or activity constraint, future variable domains have to be restored before search continues with trying a new value for  $var$ .

### 3.4 Maintaining Arc and Activation Consistency

Local consistency enforced by forward checking filters the domains of future variables directly connected through compatibility constraints to the currently instantiated variable. This level of consistency can be extended to arc consistency over all future variables, that is, both directly and indirectly connected via constraints to the current instantiation node in the search tree.

Combining backtrack search with arc consistency has resulted in the most effective solving algorithm for standard binary CSPs, Maintaining Arc Consistency (MAC) (Sabin, D. & Freuder 1994), (Grant & Smith 1995), (Bessiere & Regin 1996). The idea of MAC is that with each variable instantiation, all the other values left in that variable's domain are eliminated, and all future variables are made arc consistent. If value  $v$  is assigned to variable  $V$ , removing all the other values in the domain of  $V$  may leave without support values in future variables which are directly connected via constraints with  $V$ . The removal of not supported values in future variables may determine a “domino effect” of further value elimination through constraint propagation in the future variable space. If arc consistency checking leads to wiping out some future variable domain, then assigning  $v$  to  $V$  fails and

**Algorithm 3.7.** *Forward checking algorithm for solving conditional CSPs. The code enclosed in boxes shows the differences between forward checking search and backtrack search as described in Algorithm 3.3.*

```

boolean CCSP_SolveFc( ) {
    Agenda  $\leftarrow$  initially included variables
    if ( Agenda is empty )
        return false
    numberSolutions  $\leftarrow$  0
    oneSolutionOnly  $\leftarrow$  whether one or all solutions are searched for
    return CCSP_Fc, oneSolutionOnly
} //end CCSP_SolveFc()

boolean CCSP_Fc(Agenda, oneSolutionOnly) {
    if ( Agenda is empty ) {
        numberSolutions  $\leftarrow$  numberSolutions + 1
        return true
    }
    var  $\leftarrow$  select variable and remove from Agenda
    value  $\leftarrow$  select value from domain of var and instantiate var
    UndoActivity  $\leftarrow$   $\emptyset$ 
    UndoValues  $\leftarrow$   $\emptyset$ 

    if ( FcCompatibility(var, value, UndoValues) and
        FcActivity(var, value, Agenda, UndoValues, UndoActivity) ) {
        if ( CCSP_Fc(Agenda, oneSolutionOnly) and oneSolutionOnly ) {
            restore all removed values saved in UndoValues

            reset variable activity status as saved in UndoActivity
            unstantiate var and put it back into the Agenda;
            return true
        }
    }

    restore all removed values saved in UndoValues
    reset variable activity status as saved in UndoActivity
    unstantiate var and put back it back into the Agenda
    UndoValues  $\leftarrow$  {(var, value)}
    remove value from domain of var
    if ( domain of var is empty )
        backtrackSearch  $\leftarrow$  false
    else
        backtrackSearch  $\leftarrow$  CCSP_Fc(Agenda, oneSolutionOnly)
    reset variable activity status as saved in UndoActivity
    restore all removed values saved in UndoValues
    return backtrackSearch
} //end CCSP_Fc()

```

MAC has to undo all value removals, remove  $v$  from  $V$ , reestablish arc consistency, and continue the search.

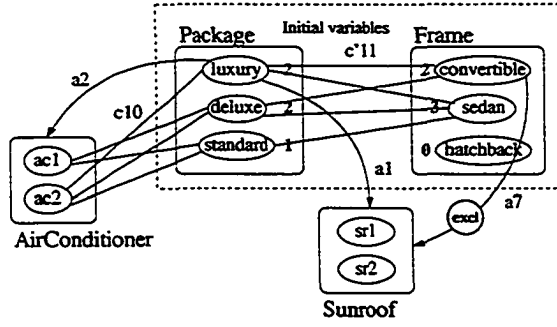
MAC algorithm for conditional CSPs has a preliminary arc consistency processing over all initial variables before the actual search starts. Arc consistency processing has received constant attention in the research community over the past 15 years, during which seven algorithms, AC-1 to AC-7, have been developed. A useful review of the latest developments can be found in (Bessière & Régin 1997; 2001).

It is important to note that although research has addressed the issue of generalizing arc consistency to  $n$ -ary constraints (Bessière & Régin 1997; Bessière 1999; Bessière *et al.* 2002), no implementation of MAC for standard non-binary constraints has been presented yet. In developing a MAC algorithm for conditional CSPs we restrict the problem to a conditional binary CSP, where both compatibility and activity constraints are binary. Under this assumption, activity constraints have unary condition constraints. In the rest of this section, the term constraints refers to binary constraints.

### 3.4.1 Example

The implementation of MAC we propose for conditional CSPs is based on AC-4 algorithm for enforcing arc consistency. The underlying mechanism of AC-4 uses support counters to keep track of the amount of support a value has along a constraint from other participating domains. Each constraint has support counters for each value in the constraint's variables. Given a binary constraint defined on variables  $X$  and  $Y$ , there is a *support counter* for each value  $x$  in domain of  $X$  such that it shows how many values in domain of  $Y$  are consistent with  $x$ . To understand the support counter concept and how it is applied to maintaining arc consistency we give two examples. The first example introduces the concept and shows how support counters are computed and then used to make a problem arc consistent. The second example is used to trace the search algorithm, and it shows how support counters are updated in the process of maintaining compatibility and activation consistency.

**Example 6.** The sample problem for which we traced backtrack and forward checking search is modified by adding a third value, *hatchback*, in the domain of variable *Frame* (Figure 3-iv).  $c'_{11}$  modifies the original  $c_{11}$  to show that *Package* has no support for the



$$\begin{aligned}
 a_1 : \text{Package} = \text{luxury} &\xrightarrow{\text{incl}} \text{Sunroof} \\
 a_2 : \text{Package} = \text{luxury} &\xrightarrow{\text{incl}} \text{AirConditioner} \\
 a_7 : \text{Frame} = \text{convertible} &\xrightarrow{\text{excl}} \text{Sunroof}
 \end{aligned}$$

$$c_{10}^{\text{allowed}} : \{(\text{luxury } ac2)(\text{deluxe } ac1)(\text{deluxe } ac2)(\text{std } ac1)(\text{std } ac2)\}$$

$$c_{11}^{\text{allowed}} : \{(\text{luxury convertible})(\text{luxury sedan})(\text{deluxe convertible})(\text{deluxe sedan})(\text{standard sedan})\}$$

Figure 3-iv: New value, *hatchback*, is added to the sample problem in Example 3.  $c'_{11}$  is modified to show that *Package* has no support for *hatchback*. Values participating in compatibility constraints have associated support counters.

newly added value *hatchback*. Compatibility constraints  $c_{10}$  and  $c'_{11}$  are expressed as sets of allowed value pairs. This representation facilitates the computation of support counters for all values participating in these two compatibility constraints.

For example,  $c'_{11}$ , defined on *Package* and *Frame*, has exactly two allowed value combinations in which value *luxury* participates :  $(\text{luxury}, \text{convertible})$  and  $(\text{luxury}, \text{sedan})$ . Therefore, on  $c'_{11}$ , *luxury* has a support counter of 2. Similarly, we compute all support counters for the domains of variables *Package* and *Frame*, as constrained by  $c'_{11}$ . The support value of 0 for *hatchback* on constraint  $c'_{11}$  shows that *hatchback* is not consistent with any other value at *Package* and can be removed from the domain of *Frame*. Only  $c'_{11}$  imposes restrictions on the value combinations of the initial variables in our sample problem.

△

The preliminary arc consistency for the example problem produces the support counters shown in Figure 3-iv. When applied to conditional CSPs, the arc consistency preprocessing phase for making the problem arc consistent considers only initial variables (the only ones that are active) and is carried out along compatibility constraints on these variables. During

search, active variables continue to be maintained arc consistent with regard to compatibility constraints. We also refer to this type of consistency as *compatibility consistency*. In addition, a special local consistency is enforced among active variables along activity constraints. We call it *activation consistency*.

The following example shows how local consistency, in its two forms, arc or *compatibility consistency* and *activation consistency*, is combined with backtrack search in a new MAC algorithm for solving conditional CSPs.

**Example 7.** We start by assigning *convertible* to *Frame* and eliminating the other value left in the domain, *sedan* (as shown at the top of the Figure 3-v). Value *hatchback* was already eliminated by the preliminary arc-consistency over the initial variables. Along the compatibility constraint  $c'_{11}$ , *sedan* supports all three values at *Package* (Figure 3-iv). The elimination of *sedan* is propagated through  $c'_{11}$  and support counters of *luxury*, *deluxe*, and *standard* are decremented. Consequently, *standard*'s support counter becomes 0. This indicates that *standard* value is inconsistent, cannot extend  $Frame = convertible$  to a partial solution, and must be removed. No more propagation takes place at this point, since no other compatibility constraints are defined on present search variables, *Frame* and *Package*.

Following the checking of compatibility constraints, we consider the activity constraints whose condition variables are active.  $a_7$  is the only such activity constraint. It is satisfied by  $Frame = convertible$ , and *Sunroof*, a variable new to the current search, is marked as excluded from the search tree rooted at *convertible*. However, there is another activity constraint,  $a_1$ , whose condition involves future variable *Package*, and which conflicts with  $a_7$ . Therefore,  $a_1$ 's condition value *luxury* is inconsistent with *convertible*, so it gets removed too from the domain of *Package*. This removal propagates on  $c'_{11}$  and decrements *convertible*'s counter to 1. With this level of local consistency achieved, we continue the search and instantiate *Package* with the only value left in its domain, *deluxe*. There is a single constraint to be checked,  $c'_{11}$ . It holds and we obtain the first solution to the problem: (*convertible*, *deluxe*).

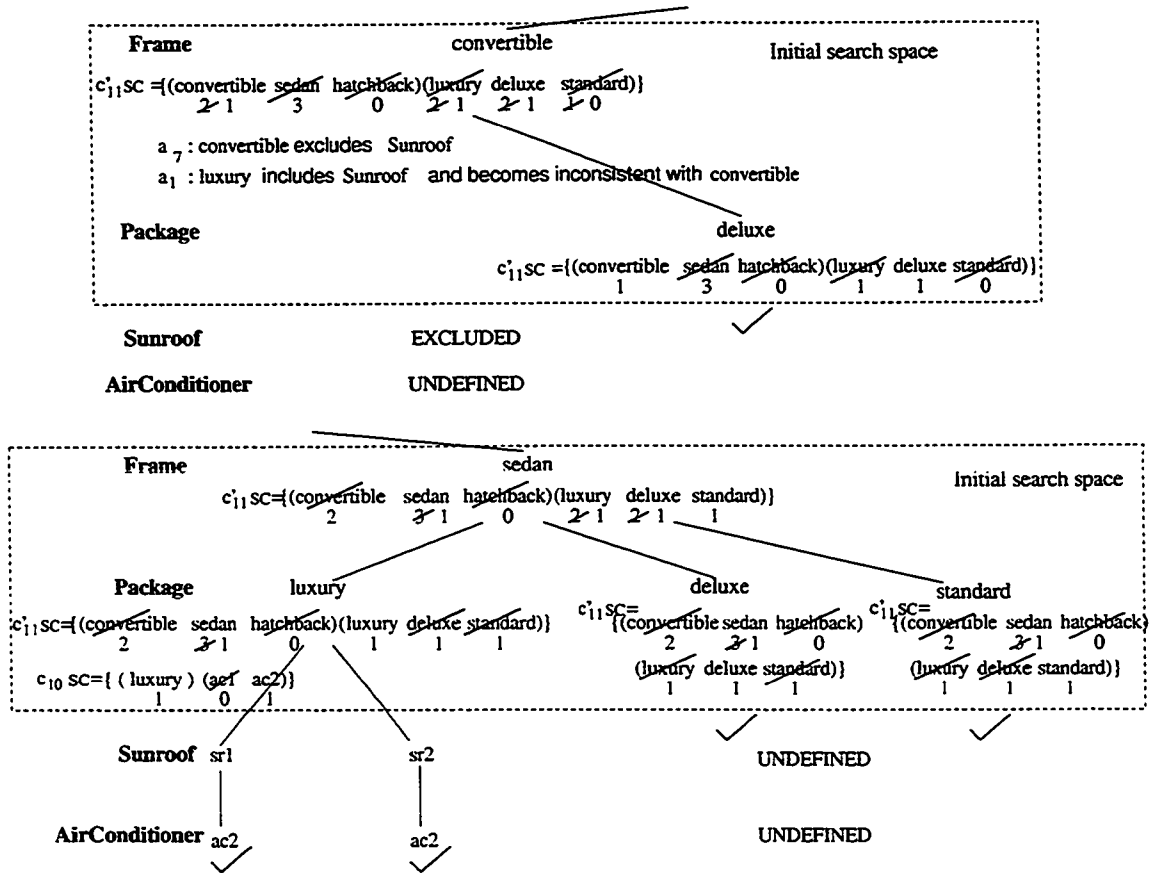


Figure 3-v: MAC search trace of the sample problem in Example 6.  $c_{10}sc$  and  $c'_{11}sc$  list the support counters computed for  $c_{10}$  and  $c'_{11}$  constraints.

To find all solutions, we back up one level and try *sedan* for *Frame* (as shown at the bottom of the Figure 3-v), remove *convertible* from *Frame*'s domain, and propagate this value removal along  $c'_{11}$  by decrementing accordingly the support counters for *Package*'s values: all become 1. There is no other constraint participating in maintaining arc consistency for *Frame* = *sedan* assignment, and we proceed by trying *luxury* for *Package*. The other two values, *deluxe* and *standard*, are removed, but no more counter updates or value removals take place along the only applicable compatibility constraint  $c'_{11}$ .

Activity constraints are checked next.  $a_1$  and  $a_2$  have *Package* = *luxury* as condition. The search path satisfies both, and two new variables, *Sunroof* and *AirConditioner* are added to the search. To make the new variables arc consistent with the active vari-

ables *Frame* and *Package*, we consider compatibility constraints between new variables and active variables. There is one such compatibility constraint,  $c_{10}$ , between the new variable *Sunroof* and active variable *Package*. We first compute  $c_{10}$ 's support counters. At this point during search, *Package* has one value left, *luxury*, with a single support at *AirConditioner*, from *ac2*. The support counters of *ac1* and *ac2* are 0 and 1: *ac1* has no support because both *deluxe* and *standard* are removed, and *ac2* has one support from *luxury*. With the removal of *ac1* value, *AirConditioner* is made arc consistent with the rest of the variables, and no further propagation takes place. The instantiations of *Sunroof* and *AirConditioner* produce two more solutions,  $(\text{sedan}, \text{luxury}, \text{sr1}, \text{ac2})$  and  $(\text{sedan}, \text{luxury}, \text{sr2}, \text{ac2})$ . We complete the search in a similar fashion by backing up one level and trying, in turn, *deluxe* and *standard* for *Package*. Each assignment extends  $\text{Frame} = \text{sedan}$  with a solution:  $(\text{sedan}, \text{deluxe})$  and  $(\text{sedan}, \text{standard})$ .

△

### 3.4.2 Algorithm

The conditional analogue of standard MAC interleaves backtrack search with maintaining consistency for compatibility constraints and activity constraints. In the following, as in forward checking for conditional CSPs, we first examine propagation of compatibility constraints to achieve arc consistency, as described in *MacCompatibility* procedure. We then extend this level of local consistency with the propagation of activity constraints (*MacActivity* procedure). The order in which constraints are checked in the conditional MAC algorithm is the same as in backtrack search and forward checking: compatibility constraints are checked first, followed by activity constraints. This order reduces the amount of undo operations caused by restoring removed values and variable activity status in case a constraint check fails. During activity constraint checking, newly included and excluded variables are listed in a new structure that is used for enforcing local consistency along activity constraints (*MacNewvar* procedure). All these procedures are put together in *CCSP\_SolveMac* algorithm, which describes conditional MAC search.

## Maintaining arc consistency over compatibility constraints

*MacCompatibility* procedure (Algorithm 3.8) takes a list of pairs of variables with their associated values, which have been removed from the variable domains,  $(V, removedValue)$ .

We denote this list *UndoValues*.

**Algorithm 3.8.** *Maintain arc consistency over compatibility constraints: future variables in the Agenda are pruned of values that are not arc consistent.*

```

boolean MacCompatibility(UndoValues, Agenda) {
  for each ( (V, removedValue) pair saved in UndoValues ) {
     $C \leftarrow$  (binary) compatibility constraints define on V and othervar, othervar  $\in$  Agenda
    for each ( c  $\in C$  ) {
      SupportValues  $\leftarrow$  values in othervar supported by removedValue of V
      for each ( sv  $\in$  SupportValues ) {
        counter  $\leftarrow$  sv's support counter along c
        save counter information in UndoValues
        counter  $\leftarrow$  counter - 1 // sv lost support from removedValue along c
        if ( counter is 0 ) {
          if sv is the only value left in the domain of othervar
            return false
          remove sv from the domain of othervar
          UndoValues  $\leftarrow$  UndoValues  $\cup$  (othervar, sv)
        }
      } // end of SupportValues list
    } // end for each c
  } // end of UndoValues list
  return true
} // end MacCompatibility()

```

Arc consistency over compatibility constraints is done for each constraint *c* that involves a variable *V* in the pair  $(V, removedValue)$ . Constraints are binary and connect *V* with a second variable, *othervar*, which is a future variable managed by the *Agenda* data structure. The procedure determines all *sv* values at *othervar* which support *removedValue* at *V*. Each *sv* value has a support counter, called *counter*, that indicates how many values support *sv* along *c*. Because *removedValue* was removed from *V*, each *sv* value loses one support value and its *counter* is decremented. If *counter* becomes 0, *sv* is left with no support at *V*, violates *c*, so it is, in turn, removed. If *sv* removal wipes out the domain of *othervar*, *MacCompatibility* fails and returns false. Otherwise, we safely remove *sv* and

add  $(othervar, sv)$  pair to the *UndoValues* list. Arc consistency processing continues until the end of the list *UndoValues* is reached. That is when *MacCompatibility* succeeds.

Anytime in the process, if *MacCompatibility* fails, variable domains modified by arc consistency are restored based on undo information saved in *UndoValues*. To avoid the recomputation of counters, for each removed value processed, *MacCompatibility* saves counter references to counters that are decremented. If the removed value is to be restored, so are the counters that that removed value has modified.

The new algorithm, *CCSP\_SolveMac*, uses *MacCompatibility* in several situations, all of which have in common some value removals. One situation is when the initial variables are made arc consistent, by calling *MakeAC* procedure, prior to launching backtrack search (see Algorithm 3.9). Before *MacCompatibility* is called, each compatibility constraint on active variables is processed so that values with no support in the domains of constraint variables are eliminated. The procedure *MakeOneAC* removes arc inconsistent values and saves them in the *UndoValues* structure. The value removals are propagated with *MacCompatibility* in order to make the initial problem arc consistent.

Another situation in which *MacCompatibility* is called by the solving method is when a variable *var* is instantiated with some *value* during search. All the values in the domain of *var* except *value* are removed and saved in the *UndoValues* list. This initial *UndoValues* list triggers arc consistency by propagating compatibility constraints that involve *var* and future variables. In the process, values at future variables might get removed and information about those variables and their removed values is added to the *UndoValues* list. The propagation continues over compatibility constraints that involve future variables until no value removal takes place and the end of *UndoValues* is reached.

If the current instantiation,  $var = value$ , fails, *value* is removed from the domain of *var*. This removal is another case in which *MacCompatibility* is called to establish arc consistency.

Finally, we will see next that maintaining local consistency over activity constraints causes value removals and, consequently, requires *MacCompatibility* propagation.

**Algorithm 3.9.** *Make initial variables arc consistent. Call MakeOneAC procedure to remove arc inconsistent values from the domains of all compatibility constraint variables. Call MacCompatibility to propagate value removal and achieve arc consistency.*

```

boolean MakeAC(Agenda) {
    UndoValues  $\leftarrow \emptyset$ 
    for each (constraint  $c$  defined on  $X$  and  $Y$  variables in the Agenda)
        if ( MakeOneAC( $c, X, Y, UndoValues$ ) is false )
            return false
    return MacCompatibility(UndoValues, Agenda)
} //end MakeAC()

boolean MakeOneAC( $c, Xvar, Yvar, UndoValues$ ) {
    set up AC-4 counters on constraint  $c$ 
    for each (  $xval$  in domain of  $Xvar$  with no support in domain of  $Yvar$  on  $c$  )
        if (removeValue( $xval, Xvar, UndoValues$ ) is false )
            return false
    for each (  $yval$  in domain of  $Yvar$  with no support in domain of  $Xvar$  on  $c$  )
        if (removeValue( $yval, Yvar, UndoValues$ ) is false )
            return false
    return true
} //end MakeOneAC()

boolean removeValue( $value, var, UndoValues$ ) {
     $D_{var} \leftarrow$  domain of  $var$ 
    if ( ( $D_{var} - value$ ) is empty )
        return false
    remove  $value$  from  $D_{var}$ 
    UndoValues  $\leftarrow UndoValues \cup (var, value)$ 
    return true
} //end removeValue()

```

### Maintaining activation consistency over activity constraints

Activity constraints are checked and propagated with *MacActivity* procedure. This procedure is called by *CCSP\_SolveMac* following the propagation of compatibility constraints, which leaves the problem arc consistent when a new *value* is tried for current variable *var*. The procedure *MacActivity* in Algorithm 3.10 shows the implementation of checking the activity constraints relevant to variable assignment  $var = value$ . If these activity constraints are satisfied, variables new to the search are made consistent over compatibility and activity constraints by calling *MacNewvar* for each such variable.

*MacActivity* implementation (Algorithm 3.10) is similar to *FcActivity* implementation

**Algorithm 3.10.** *Maintaining condition consistency over activity constraints. The code enclosed in boxes differentiates MacActivity from BtActivity, Algorithm 3.2. It differs from FcActivity in two ways (see nested boxes): newly excluded targets are added too to the NewVariables list, and MacNewvar is called to make NewVariables arc and condition consistent.*

```

boolean MacActivity(var, value, Agenda, UndoValues, UndoActivity) {
   $A \leftarrow$  (binary) activity constraints whose (unary) conditions involve variable var
  for each ( $a \in A$ ) {
    if (value is not consistent with a's condition)
      return true //activation condition fails; no effect on variables' activity status
    else {
      target  $\leftarrow$  target variable of a
      action  $\leftarrow$  activity performed by a
      if (action includes target) {
        if (target has already been excluded)
          return false //conflicting activity constraints
        if (target is newly included) {
           $Agenda \leftarrow Agenda \cup target$  //target becomes future (active) variable
           $NewVariables \leftarrow NewVariables \cup target$ 
        }
      }
    }
    else { // action excludes target
      if (target has already been included)
        return false //conflicting activity constraints
       $NewVariables \leftarrow NewVariables \cup target$ 
    }
     $UndoActivity \leftarrow UndoActivity \cup a$ 
  } //end else
} //end for

for each (newvar  $\in$  NewVariables) {
   $LocalUndoValues \leftarrow \emptyset$ 
  macNewResult  $\leftarrow$  MacNewvar(newvar, LocalUndoValues)
   $UndoValues \leftarrow UndoValues \cup LocalUndoValues$ 
  if (macNewResult is false)
    return false
}

return true
} //end MacActivity()

```

(Algorithm 3.5). One important difference, which is true of *MacCompatibility* too, is the assumption that constraints are binary. Thus, activity constraints have unary conditions of the form  $V = v$ . To facilitate the comparison among *MacActivity*, *FcActivity*, and

*BtActivity*, we boxed that portion of the code that distinguishes simply checking the activity constraints, as shown in *BtActivity* (Algorithm 3.2) and used in the backtrack algorithm, *CCSP\_Bt*, from combining it with some form of local consistency initiated over the newly activated variables, *NewVariables*, as shown in *FcActivity* (Algorithm 3.5). A nested set of boxes mark updates and additions specific to *MacActivity*.

We note that *MacActivity*, unlike *FcActivity*, collects in *NewVariables*, along with the newly included targets, the target variables which have been excluded for the first time. Second, while *FcActivity* uses *FcNewvar* to filter the domains of the newly included variables, *MacActivity* calls *MacNewvar* to make the new problem, extended with new variables, consistent over both compatibility and activity constraints. No value is removed when activity constraints are checked in *MacActivity*. That is why *MacNewvar* is called with an empty list of removed values. During its execution, *MacNewvar* propagates activity constraints and eliminates condition values that contradict activity status of problem variables. Next, we present the implementation of the *MacNewvar* procedure in Algorithm 3.11.

The *MacNewvar* procedure has two cases, depending on the activity status of *newvar* as either excluded or included. The first case is when *newvar* is excluded from the search space. This activity status makes inconsistent values at future condition variables of inclusion activity constraints, which have not been processed yet, but which target *newvar*. Propagation on those activity constraints leads to condition value removal. *UndoValues* collects these removed values.

The second case is when *newvar* is included, that is, made active. Local consistency in this case regards both compatibility and activity constraints. *newvar* has to be arc consistent with all the other active variables with which it shares compatibility constraints. Therefore, *MakeOneAC* procedure is called on each such compatibility constraint to remove values with no support in *newvar* and its neighboring variables. The removed values are added to the *UndoValues* list.

As an active variable, *newvar* might participate in two different roles in activity constraints which have not been processed yet: as condition variable and/or target variable.

**Algorithm 3.11.** *Maintain arc consistency with a newly included or excluded variable newvar over both compatibility and activity constraints defined on active variables.*

```

boolean MacNewvar(newvar, Agenda, UndoValues) {
  if ( newvar is newly excluded ) {
    InclTarget  $\leftarrow$  activity constraints that include newvar as target
    for each ( it  $\in$  InclTarget, where it : CondVar = val  $\xrightarrow{\text{incl}}$  newvar
               and CondVar is on the Agenda)
      if (removeValue(val, CondVar, UndoValues) is false)
        return false
  }
  else { // newvar is newly included
    C  $\leftarrow$  compatibility constraints involving newvar and othervar, with othervar active
    for each ( c  $\in$  C)
      makeOneAC(c, newvar, othervar, UndoValues)
    ExclTarget  $\leftarrow$  activity constraints which exclude target newvar
    for each ( et  $\in$  ExclTarget, where et : CondVar = val  $\xrightarrow{\text{excl}}$  newvar
               and CondVar is on the Agenda)
      if (removeValue(val, CondVar, UndoValues) is false)
        return false
    SourceAct  $\leftarrow$  activity constraints whose source is newvar
    for each ( sa  $\in$  SourceAct, such that
               either sa includes some target variable which is already excluded
               or sa excludes some target variables which is already included in the Agenda)
      if (removeValue(condition, newvar, UndoValues) is false)
        return false
  }
  return MacCompatibility(UndoValues, Agenda)
} //end MacNewvar()

```

As a condition variable, *newvar* can participate in either (1) an inclusion activity constraint that targets an excluded variable, (2) or in an exclusion activity constraint that targets an included variable. In either situation, *newvar*'s condition value is inconsistent with these activity constraints. As a target variable, since *newvar* status is included, it can render inconsistent condition values of exclusion activity constraints. Inconsistent values are removed and saved in the *UndoValues* list.

The last step in *MacNewvar* is when all value removals saved in the *UndoValues* due to the elimination of inconsistent values over activity and compatibility constraints are propagated with *MacCompatibility*. If the propagation is successful, *newvar* is made consistent with the problem variables on both types of constraints.

### *CCSP\_Mac* algorithm

*CCSP\_SolveMac* in Algorithm 3.12 uses the recursive function *CCSP\_Mac*, which implements MAC for solving conditional CSPs. Unlike *CCSP\_SolveBt* (Algorithm 3.3), *CCSP\_SolveMac* makes arc consistent compatibility constraints on all initial variables, by calling *MakeAC* procedure.

When *CCSP\_Mac* instantiates a variable *var* with some *value* in its domain, all the other values are removed from the domain and saved in the *UndoValues* list. With this list, *MacCompatibility* initiates the arc consistency processing, during which possibly more values are removed and added to the *UndoValues* list. After all arc inconsistent values are removed from the *Agenda*'s variables, without wiping out any of their domains, *MacActivity* is called to check the activity constraints and eliminate those condition values which introduce inconsistency among activity constraints. When this checking succeeds, *CCSP\_Mac* is called recursively to find either one or all solutions to the problem.

Algorithm 3.12 has a very similar design to *CCSP\_SolveFc* algorithm in Algorithm 3.7. The differences are shown in a the nested set of boxes that contain algorithmic descriptions specific to *CCSP\_SolveMac*. Thus, *CCSP\_SolveMac* calls *MakeAC* prior to starting search to make the initial variables arc consistent. The recursive procedure *CCSP\_Mac* selects a variable *var* and instantiates it with a *value* as *CCSP\_Bt* and *CCSP\_Fc* do, but removes all the other values from the domain of *var* and adds pairs of *var* and removed value to *UndoValues* list. Compatibility constraints are propagated with *MacCompatibility* to reestablish arc consistency among *var* and future variables. Upon successful return of this procedure, *MacActivity* is called to check activity constraints and make possibly new variables added to the search space consistent with regard to compatibility and activity constraints. If no activity constraint fails and the new variables do not cause the elimination of all values at some future variables, *CCSP\_Mac* is called recursively to find value assignments to the rest of the problem variables.

In case there is a failure in trying *value* for *var*, or all solutions are sought, the search space and variables status have to be restored and all changes recorded in *UndoValues* and

*UndoActivity* have to be undone. The current instantiation *value* is marked as tried and removed from the domain of *var*. If other values are left in *var*'s domain, *value* removal is propagated first with *MacCompatibility* to check whether future variables remain arc consistent with the rest of values at *var*. If that is the case, a recursive call to *CCSP\_Mac* continues the search.

**Algorithm 3.12.** *Maintaining arc consistency algorithm for solving conditional CSPs. The code in the nested boxes show the differences between maintaining arc consistency search and forward checking search as described in Algorithm 3.7.*

```

boolean CCSP_SolveMac( ) {
    Agenda ← initial variables
    if ( Agenda is empty )
        return false

if ( MakeAC( Agenda ) is false )
            return false



    numberSolutions ← 0
    oneSolutionOnly ← whether one or all solutions are searched for

return CCSP_Mac(Agenda, oneSolutionOnly)


} //end CCSP_SolveMac()

boolean CCSP_Mac(Agenda, oneSolutionOnly) {
    if ( Agenda is empty ) {
        numberSolutions ← numberSolutions + 1
        return true
    }
    var ← select variable and remove from Agenda
    value ← select value from domain of var and instantiate var
    UndoActivity ← ∅

remove all values from domain of var except value
        UndoValues ← pairs (var, v), where v is var's removed value
        if ( MacCompatibility(UndoValues, Agenda) and
            MacActivity(var, value, Agenda, UndoValues, UndoActivity) ) {
            if ( CCSP_Mac(Agenda, oneSolutionOnly) and oneSolutionOnly ) {
                restore all removed values saved in UndoValues
            }
            reset variable activity status as saved in UndoActivity
            unstantiate var and put it back into the Agenda
            return true
        }



restore all removed values saved in UndoValues
        reset variable activity status as saved in UndoActivity
        unstantiate var and put it back into the Agenda
        UndoValues ← {(var, value)}
        remove value from domain of var
        if ( domain of var is empty )
            backtrackSearch ← false



else if ( not (MacCompatibility(UndoValues, Agenda) ) )
            backtrackSearch ← false
        else
            backtrackSearch ← CCSP_Mac(Agenda, oneSolutionOnly)



    reset variable activity status as saved in UndoActivity
    restore all removed values saved in UndoValues
    return backtrackSearch
} //end CCSP_Mac()

```

### 3.5 Summary

In this chapter we presented new algorithms for directly solving conditional CSPs. They adapt standard local consistency of forward checking and maintaining arc consistency to the conditional domain.

We started with a modified version of the backtrack search algorithm, *CCSP\_SolveBt*, that handles directly both types of activity constraints. Two new algorithms were derived from it:

- Forward checking, *CCSP\_SolveFc*, propagates compatibility constraints to the problem active variables.
- Maintaining arc consistency, *CCSP\_SolveMac*, propagates compatibility constraints to achieve arc consistency. This level of consistency is extended with the propagation of activity constraints to achieve activation consistency.

In the next chapter these algorithms are evaluated experimentally on large and diverse testbeds of random conditional CSPs. The experimental studies show that maintaining arc consistency outperforms forward checking, which, in turn, outperforms backtrack search. In addition, the experimental analysis allows for comparing the numbers of solutions reported by the three algorithms when run on the same problem instances. These comparisons show that all algorithms produce the same number of solutions for the same problems.

## CHAPTER 4

# EXPERIMENTAL EVALUATION

### 4.1 Introduction

The three algorithms for solving conditional CSPs, plain backtracking, forward checking, and maintaining arc consistency, were tested in experiments covering diverse populations of randomly generated problems. The experimental analysis has two objectives:

- Provides evidence about the relative efficiency of the algorithms,
- Provides some level of reassurance as to the algorithms' correctness.

Two types of studies made up the experimental evaluation. In the first category of experimental studies we measured execution time. We ran experiments for each of the three algorithms to find minimal solutions, that is, solutions that include the minimum number of active variables.

In the second category we focus on probing counters that are typically representative of algorithm effort: number of backtracks and compatibility checks for standard CSPs, and some new measurements specific to conditional CSP, such as number of condition checks, included and excluded variables, and redundant and conflicting activations. In this study the algorithms were run to find all solutions, not only the minimal sets of active variables that satisfy all constraints. The total number of solutions was reported for each of the three algorithms. We checked that the number of solutions was the same for all three algorithms when run on the same problem instances.

The algorithms were implemented in C++ on a Red Hat 8.0 distribution of the Linux platform. The experiments were run on a PC with an AMD processor at 1,200 MHz and 512MB of RAM.

## 4.2 Design

### 4.2.1 Measurements and Problem Topologies

The two basic measurements we used in our experiments are: the overall execution time and number of backtracks. In the case of backtrack search and forward checking we also counted the number of compatibility checks. The overall execution time is computed from time stamps obtained through operating system calls to obtain the current time that are placed before and after the program statement that calls the search algorithm.

The backtracks counter is incremented each time the search procedure exhausts all value assignments for the current variable, that is, its domain becomes empty. Compatibility constraints are checked for consistency in backtrack search and forward checking algorithms each time a partial solution is extended with the current variable instantiation. The compatibility check counter is incremented in the for loop of the *BtCompatibility()* (Algorithm 3.1) and *FcCompatibility()* (Algorithm 3.4) procedures.

The effort counting measurements were enhanced with other evaluation devices that measure algorithm performance with regard to problem conditional characteristics. Before we enumerate the measurements that gauge this type of algorithm effort, we recall from Chapter 2 the parameters required to generate random binary conditional CSPs. Five *standard parameters* describe standard CSP characteristics: (1) problem size as the number of variables,  $n$ , (2) maximum domain size,  $d_{max}$ , (3) compatibility density or actual number of compatibility constraints, (4) compatibility satisfiability or actual number of value pairs in a binary compatibility constraint, and (5) actual domain size.

In all experiments the problem size was fixed at 10. Maximum domain size was set to 5, 8, and 10, depending on the experimental study. The actual number of values in a domain was fixed to the maximum domain size by setting the probability of generating a value domain to 1. In all experiments we varied the topology of the underlying standard CSP (compatibility constraint graph) through two probabilities for generating: compatibility constraint elements, or density,  $d_c$ , and value pairs in a compatibility constraint, or

satisfiability,  $s_c$

Nine *conditional parameters* describe the characteristics of random binary conditional CSPs. We recall that this class of problems uses unary condition constraints to form binary activity constraints. Compatibility constraints are binary too. Basically, what controls problem activity are the actual number of condition values in a domain and the actual number of target or non-initial variables in a problem. The problem generator controls the amount of activity through two probabilities, analogs of standard parameters that determine compatibility density and satisfiability. The conditional counterparts are probabilities for generating target variables in a problem and condition values in a variable domain. The actual number of target variables indicates the density of the activity graph,  $d_a$ . The actual number of condition values per domain indicates the satisfiability of the activation condition,  $s_a$ .

The actual number of included versus excluded target variables is determined by the probability of generating inclusion activity constraints,  $p_a$ . To compute the actual number of condition values per domain and target variables per problem (included and excluded targets), the random problem generator uses  $d_a$ ,  $s_a$ , and  $p_a$  probabilities along with three maximum limits parameters. Limit parameters cap condition values and target variables per condition variable domain, *maxCondPerDom* and *maxTargetPerCond*, and total condition values per problem, *totalCond*. The last category of dynamic parameters affects activity “propagation” through already activated variables, *noCondInTarget*, and activation redundancy, *noRdntDiffDom* and *noRdntSameDom*. If *noCondInTarget* is set to true, target variables are not condition variables and do not contain condition values. If we want target variables to disseminate problem activity, this parameter is set to false. Activation redundancy is caused by activity constraints that target the same variable and have either the same condition variable or different condition variables. It can be controlled by setting the boolean parameters *noRdntSameDom* and *noRdntDiffDom*. If both parameters are false, the generated problem will exhibit a larger degree of activation redundancy.

We measure algorithm effort spent with checking activity constraints and managing

included and excluded variables by counting condition constraint checks, and included and excluded variables generated during search. The procedures *BtActivity*, *FcActivity*, and *MacActivity* in Algorithm 3.2, 3.5, and 3.10 increment these counters each time a condition constraint is checked or a target variable is included or excluded.

Another important aspect specific to solving conditional CSPs is the effect redundant and conflicting activity constraints have on the overall algorithm's effort. During search, we measure and report the number of redundant constraints and conflicting constraints that an algorithm examines. These counters are computed when activity constraints are checked for consistency and they either redundantly generate activity or invalidate variable activity status.

#### 4.2.2 Experimental Studies

For all the problem sets used in our experiments, activity maximum limits were set to:

$$\begin{aligned} \text{maxCondPerDom} &= s_a * d_{\text{max}} \\ \text{totalCond} &= s_a * d_{\text{max}} * n \\ \text{maxTargetPerCond} &= n/2 \end{aligned}$$

where  $n$  is the number of variables,  $d_{\text{max}}$  is the number of values per domain (domain size), and  $s_a$  is the activity satisfiability.

All boolean dynamic parameters were set to false. The probability that an activity constraint is an inclusion, as opposed to exclusion, activity constraint,  $p_a$ , was set to 0.5.

Execution time and counting algorithm effort are the two types of studies we designed for our experiments. In the first category we had the following two studies:

**Study 1: BT and FC execution time for finding minimal solutions.** We compared the execution time, in number of seconds, for backtrack search (BT) and forward checking (FC), when looking for all solutions of minimum size. The relative performance of these two algorithms was not studied for finding all solutions for two reasons. The restriction to finding minimal solutions rather than all solutions (1) provided very conclusive results in support of FC's efficiency over BT, and (2) reduced significantly the time spent for running this experimental study.

All problems used for this experiment were the same size:  $n = 10$  variables and  $d_{max} = 8$  values per domain. The compatibility density,  $d_c$ , varied in the range  $[0.1 \dots 0.4]$  in increments of 0.02, while the compatibility satisfiability was fixed at  $s_c = 0.25$ . These two parameters characterize the topology of the underlying standard CSP of all variables, including non-initial variables, and all compatibility constraints they involve. These parameters do not, however, control the density or satisfiability of the derived conditional CSP, in which the set of active variables and, consequently, the set of compatibility constraints vary dynamically with the activity constraints. The activity satisfiability and the activity density were also fixed,  $s_a = 0.3$ ,  $d_a = 0.3$ . For each of the 16  $(d_c, s_c, d_a, s_a)$  problem classes we randomly generated 100 problems.

**Study 2: FC and MAC execution time for finding minimal solutions.** We compared the execution time, in number of seconds, for FC and maintaining arc consistency (MAC), when looking for the solutions of minimum size.

All problems used for this experiment were the same size:  $n = 10$  variables and  $d_{max} = 10$  values per domain. The compatibility density and compatibility satisfiability were fixed,  $d_c = 0.2$ ,  $s_c = 0.2$ . The activity satisfiability,  $s_a$ , and the activity density,  $d_a$ , varied in the range  $[0.1 \dots 0.9]$  in 0.1 increments. For each of the 81  $(d_c, s_c, d_a, s_a)$  problem classes we randomly generated 100 problems. Because the efficiency gain MAC shows over FC is more limited than the FC's gain over BT (as demonstrated in Study 1), we are interested in a more extensive study that (1) considers many more topological classes and (2) examines how problem conditionality influences the solving time.

Counting algorithm performance is the objective of the second type of experimental studies. They are:

**Study 3: BT and FC relative performance for finding all solutions.** We studied the relative efficiency of BT and forward checking FC algorithms when searching for all solutions. The classes of random problems on which we ran these algorithms have the same size, that is, the number of variables  $n = 10$  and domain size is  $d_{max} = 5$ .

Problem topologies were generated by varying compatibility density  $d_c$  and compatibility

satisfiability  $s_c$  in steps of 0.1 in the range of  $[0.4 \dots 0.8]$  and  $[0.1 \dots 0.5]$ , respectively. For each problem class that corresponds to a  $(d_c, s_c)$  value combination we varied conditional parameters of density and satisfiability,  $d_a$  and  $s_a$ , in steps of 0.1 in the range  $[0.1 \dots 0.9]$ , and generated 81 subclasses of 10 problem instances each. The number of inclusion activity constraints was set to the number of exclusion activity constraints, that is,  $p_a = 0.5$ .

We measured the number of backtracks, compatibility checks, and condition checks.

**Study 4: MAC and FC relative performance for finding all solutions.** We studied the relative efficiency of FC and MAC algorithms when searching for all solutions. The classes of random problems on which we ran these algorithms have size and activity maximum limits set to the same values as in Study 3. Problem topologies were generated by varying compatibility density and satisfiability in steps of 0.1, usually in the range  $[0.1 \dots 0.9]$ . For sparse underlying compatibility constraint graphs characterized by low density values of 0.1 and 0.2, compatibility satisfiability was varied in shorter ranges of  $[0.1 \dots 0.5]$  and  $[0.1 \dots 0.6]$ , respectively, to reduce the solution space and, consequently, the running time. For all the other compatibility density values in the range  $[0.3 \dots 0.9]$ , compatibility satisfiability was chosen in the range  $[0.1 \dots 0.8]$ , except for  $d_c = 0.3$  whose corresponding  $s_c$  range was  $[0.1 \dots 0.9]$ . As in Study 1,  $d_a$  and  $s_a$  were varied in steps of 0.1 in the range  $[0.1 \dots 0.9]$ . For each  $(d_c, s_c, d_a, s_a)$  problem class we generated 10 problems. We measured the number of backtracks, condition checks, number of included and excluded variables, and number of checks for redundant and conflicting activations.

## 4.3 Analysis

### 4.3.1 Execution Time

#### Study 1: BT and FC execution time for finding minimal solutions

Test Suite. We ran multiple sets of experiments, on problems of various sizes and with different value ranges for compatibility and activity parameters. The following is one snapshot which we found, based on our results, to be representative of the relation between

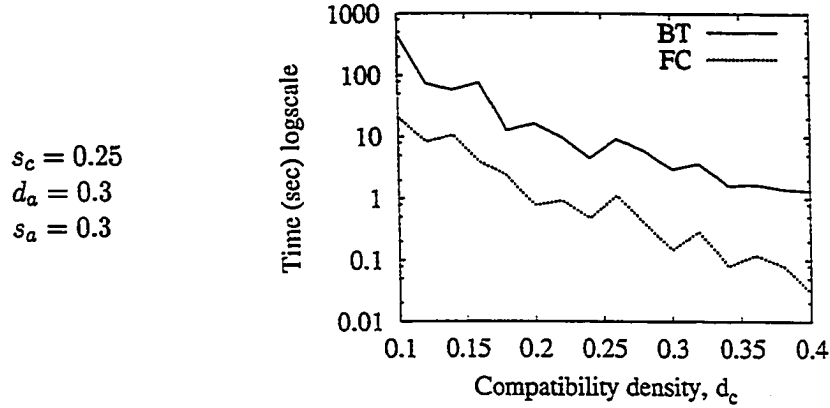


Figure 4-i: Execution time for running BT and FC as function of compatibility density,  $d_c$ , in the range  $[0.1 \dots 0.4]$  in increments of 0.02. The other three problem topology parameters are fixed:  $s_c = 0.25$ ,  $d_a = 0.3$ , and  $s_a = 0.3$ .

the execution times for BT and FC across the entire topological problem space we explored.

The problem sets for this experiment have the compatibility density,  $d_c$ , varied in the range  $[0.1 \dots 0.4]$  in increments of 0.02. Values larger than 0.4 yielded problems with execution times practically 0.

To maintain the execution time per problem within an acceptable range, usually under 10 minutes on average, we generated problems with 10 variables, each with 8 values per domain, and tried to avoid problems with very large solution sets by fixing the compatibility satisfiability  $s_c = 0.25$ .

The activity satisfiability and the activity density were also fixed,  $s_a = 0.3$ ,  $d_a = 0.3$ . For each of the 16  $(d_c, s_c, d_a, s_a)$  problem classes we generated 100 problems.

**Results.** The main observation on the data presented in Figure 4-i is that FC always outperforms BT in execution time by one order of magnitude (note the logarithmic scale).

## Study 2: FC and MAC execution time for finding minimal solutions

**Test Suite.** We explored a larger problem space than the one presented in the previous study. We conducted multiple experiments on problem sets of various sizes and with different value ranges for compatibility and activity parameters, and found the following

snapshot to be representative of the relation between execution times for FC and MAC across the entire problem space.

The test suite for this experiment consists of 81 problem classes for all  $(d_a, s_a)$  control parameter combinations, with  $d_a$  varying in  $[0.1 \dots 0.9]$  range in 0.1 increments and  $s_a$  varying in  $[0.1 \dots 0.9]$  range in 0.1 increments. In each class we generated 100 problems.

All the problems were the same size, 10 variables, each with 10 values per domain. The choice of low values for the compatibility density and compatibility satisfiability,  $d_c = 0.2$ ,  $s_c = 0.20$ , was made to avoid large solutions sets.

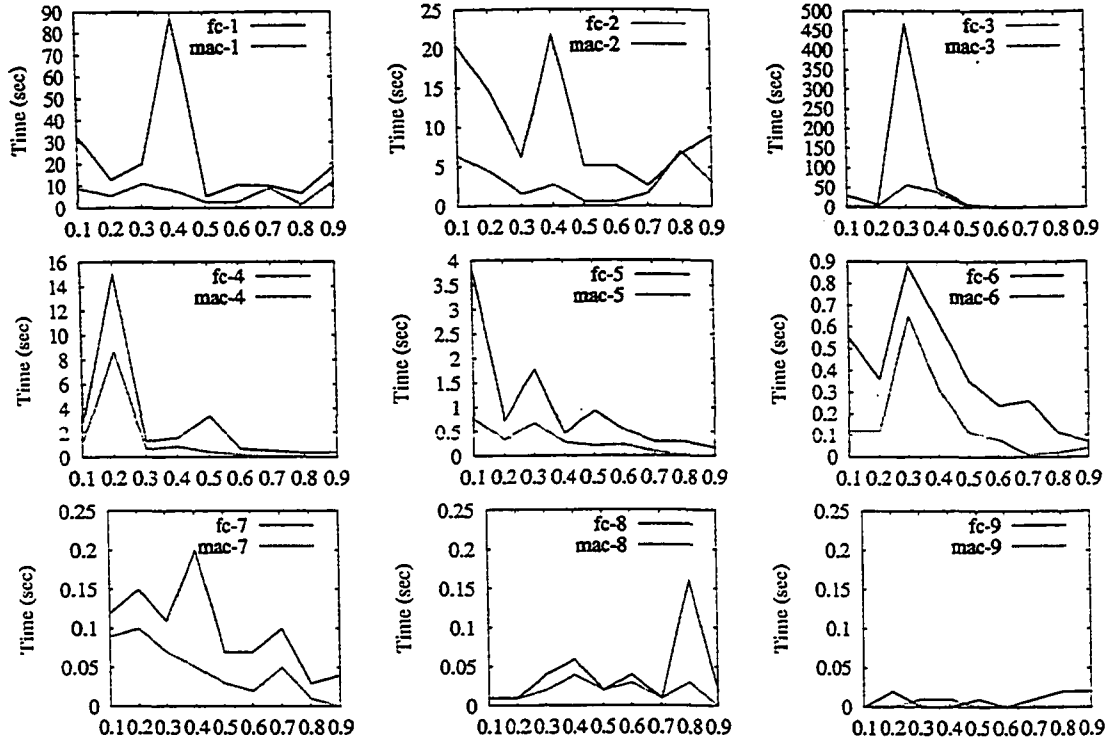
**Results.** The main result supported by the data presented in Figure 4-ii is that MAC consistently outperforms FC in execution time. Notice that the most significant gain of MAC over FC happens on the most difficult problems in the set.

### 4.3.2 Counting Effort

#### Study 3: BT and FC relative performance for finding all solutions

The test suite of this study consists of 25 problem classes for all  $(d_c, s_c)$  control parameter combinations, with  $d_c$  varying in the  $[0.4 \dots 0.8]$  range and  $s_c$  varying in the  $[0.1 \dots 0.5]$  range. In each class we generated 81 subclasses of 10 problems by varying  $d_a$  and  $s_a$  in the  $[0.1 \dots 0.9]$  range. We counted the number of backtracks, bkts, compatibility checks, compCks, and condition checks, condCks, performed on each problem, and averaged them over 10 problems in the same  $(d_c, s_c, d_a, s_a)$  topological point. There are  $5 \times 5 \times 9 \times 9 = 2,025$  classes of problems for all combinations of  $(d_c, s_c, d_a, s_a)$  we studied. We ran the two algorithms, BT and FC, measured the three effort counters, bkts, compCks, and condCks, and produced  $(2,025/9) \times 2 \times 3 = 1,350$  graphs that show effort variation with  $s_a$ .

**Example 8.** An example of this type of results is shown in Figure 4-iii. The problem topology that identifies the random problem class in this example has a compatibility density of 0.4, compatibility satisfiability of 0.1, and activity density of 0.1. We examine the variation of backtracks, compatibility checks, and condition checks counters with  $s_a$  for BT and FC (top of Figure 4-iii), and report on relative performance by computing the ratio between



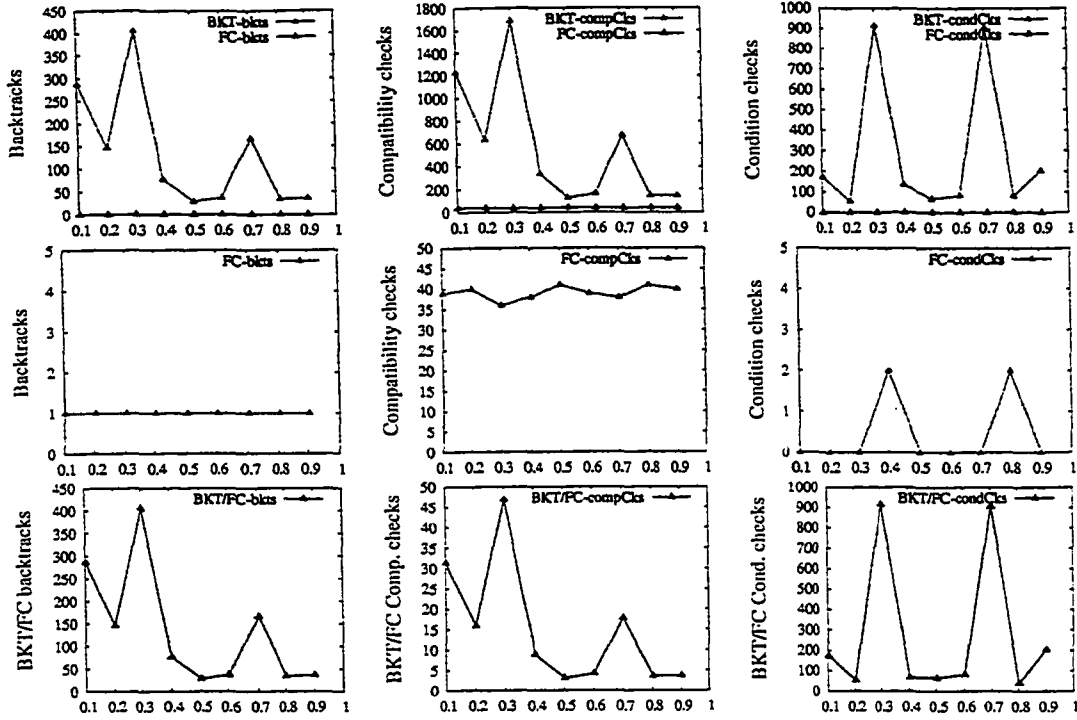
On the x-axis: satisfiability of activity,  $s_a$  (probability of generating a domain value as condition value)

Figure 4-ii: Execution for running FC and MAC as function of activity satisfiability,  $s_a$ , in the range  $[0.1 \dots 0.9]$ , in 0.1 steps. Each graph corresponds to a different density of activity,  $d_a$ , varied in the range  $[0.1 \dots 0.9]$  in 0.1 increments. The compatibility topology is fixed: compatibility density,  $d_c$ , and compatibility satisfiability,  $s_c$  are set at 0.2.

BT and FC effort for each counter<sup>1</sup> (bottom of the figure).

The main observation is that FC outperforms BT on all three measures, most significantly with regard to the condition checks. The BT/FC-condCks graph, in the lower right corner of Figure 4-iii, shows a ratio of 900 to 1 in some cases. In the case of BT, we notice that the number of backtracks and compatibility checks decreases with higher  $s_a$  values. This is caused by the tension between a low  $d_a$  and high  $s_a$  values. A  $d_a$  of 0.1 indicates that one variable out of 10 has its activity status determined by activity constraints. High  $s_a$  values designate most of the domain values as condition values which activate the same variable. That is why the number of condition checks sharply increases. The activity sta-

<sup>1</sup>To avoid division by 0, values of 0 recorded for FC performance were adjusted to 1's.



On the x-axis: satisfiability of activity,  $s_a$  (probability of generating a domain value as condition value)

Figure 4-iii: BT and FC performance measured by number of backtracks, compatibility checks, and condition checks done by each algorithm as functions of  $s_a$  for fixed  $d_c = 0.4$ ,  $s_c = 0.1$ , and  $d_a = 0.1$  (top - both BT and FC; middle - only FC). Ratios between corresponding counters show the factor by which FC outperforms BT (bottom).

tus is determined with a  $p_a$  probability of 0.5 of being included or excluded. As the same target variable is conditioned by several values to be included or excluded, more activity constraints turn to be conflicting. It means that checking activity constraints fails earlier during search and fewer backtracks and compatibility checks are done.

△

This example is just a snapshot in a large and multidimensional topological problem space. We are interested in examining whether FC outperformance holds for the larger problem topology spectrum defined by varying all four control parameters  $d_c$ ,  $s_c$ ,  $d_a$ , and  $s_a$ .

**Test Suite.** We synthesize BT-FC comparison results in three figures that show how

backtracks and the backtrack BT/FC ratio vary with  $d_c$  and  $s_a$  (Figure 4-iv),  $d_c$ , and  $d_a$  (Figure 4-v), and  $d_a$  and  $s_a$  (Figure 4-vi) for three compatibility satisfiability levels:  $s_c = 0.1$  (left column),  $s_c = 0.3$  (middle column), and  $s_c = 0.5$  (right column). The fourth control parameter in each figure was fixed at two levels, low and high, which are (0.2, 0.6) for fixed  $d_a$  in Figure 4-iv and fixed  $s_a$  in Figure 4-v, and (0.4, 0.8) for fixed  $d_c$  in Figure 4-vi. The variation of compatibility checks, condition checks, and their corresponding BT/FC ratios is shown using a logarithmic scale at the bottom of each figure for one compatibility satisfiability level,  $s_c = 0.3$  and low levels for either  $d_a = 0.2$ ,  $s_a = 0.2$ , or  $d_c = -4$  when the other two control parameters vary.

Results. The comprehensive picture depicted in these figures shows for this study that:

- FC outperforms BT on all measures and for all problem topologies that we studied.
- Backtrack effort for both FC and BT increases with larger compatibility satisfiability values (higher  $s_c$ ).
- Backtrack effort decreases with larger problem activity characterized by more condition values per domain (higher  $s_a$ ) and more targeted variables per problem (higher  $d_c$ ).
- FC is better than BT by one to two orders of magnitude on the number of backtrack, bkts and compatibility checks, compCks, measures, and up to three orders of magnitude on the number of condition checks, condCks.
- Compatibility checks effort is larger than backtrack effort, and shows the same variability with problem activity as backtrack effort.
- Contrary to backtrack effort variation with problem activity, condition checks effort increases when the problem exhibits more conditionality (higher  $d_a$  and  $s_a$ ).
- All effort measures decrease with the density of the compatibility constraints (higher  $d_c$ ).

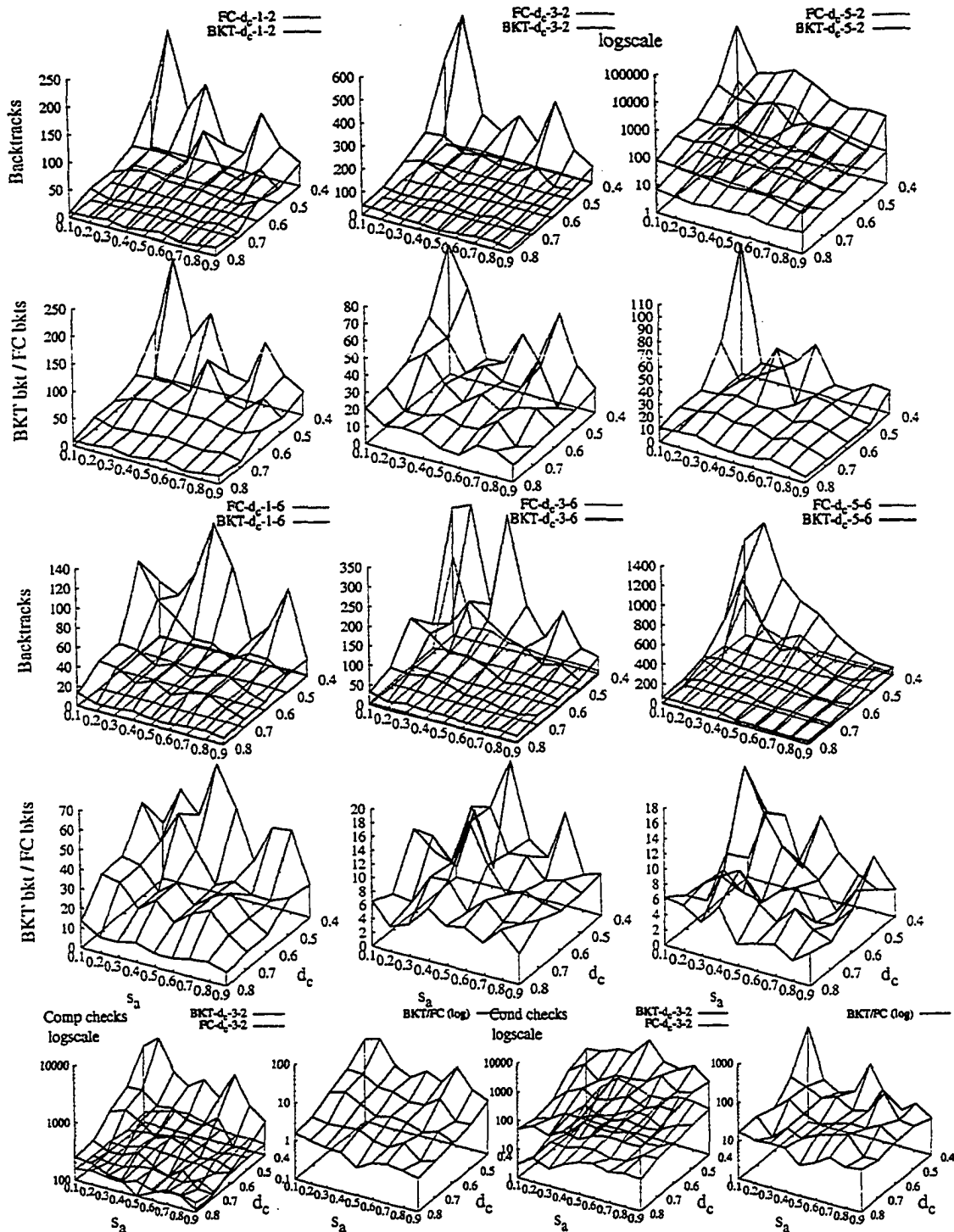


Figure 4iv: Comparison between BT and FC effort measured as the number of backtracks (rows 1 and 3), ratio of number of backtracks (rows 2 and 4), and number of compatibility and condition checks (row 5). Variation of effort with compatibility density,  $d_c$ , and activity satisfiability,  $s_a$ . Fixed activity density,  $d_a = 0.2$  (rows 1, 2, and 5) and 0.6 (rows 3 and 4). Each column corresponds to a different compatibility satisfiability value: 0.1 (left), 0.3 (middle), and 0.5 (right).

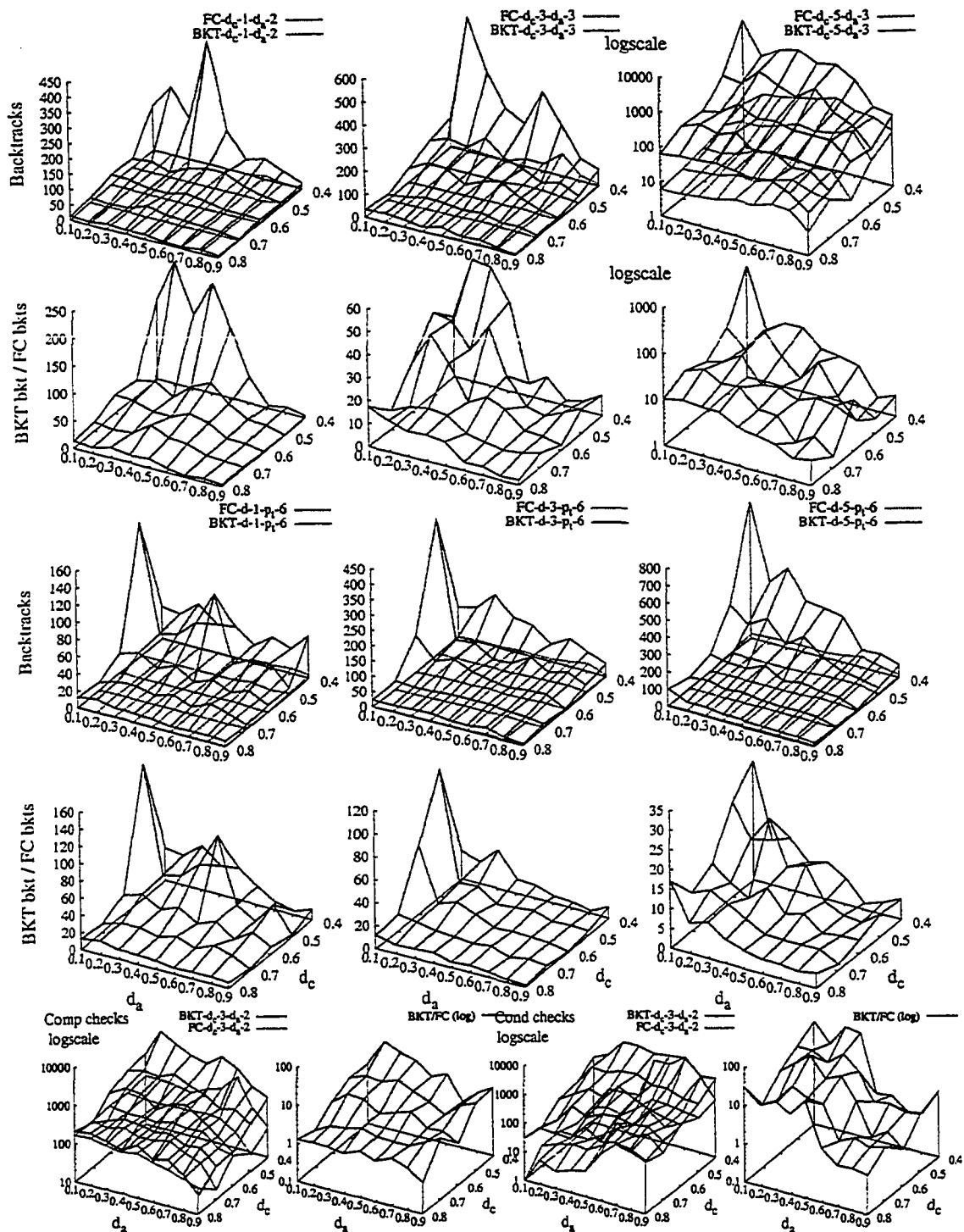


Figure 4-v: Comparison between BT and FC using the same effort measures as in Figure 4-iv. Variation of effort with compatibility density,  $d_c$ , and activity density,  $d_a$ . Fixed activity satisfiability,  $s_a$ , of 0.2 (rows 1, 2, and 5) and 0.6 (rows 3 and 4).

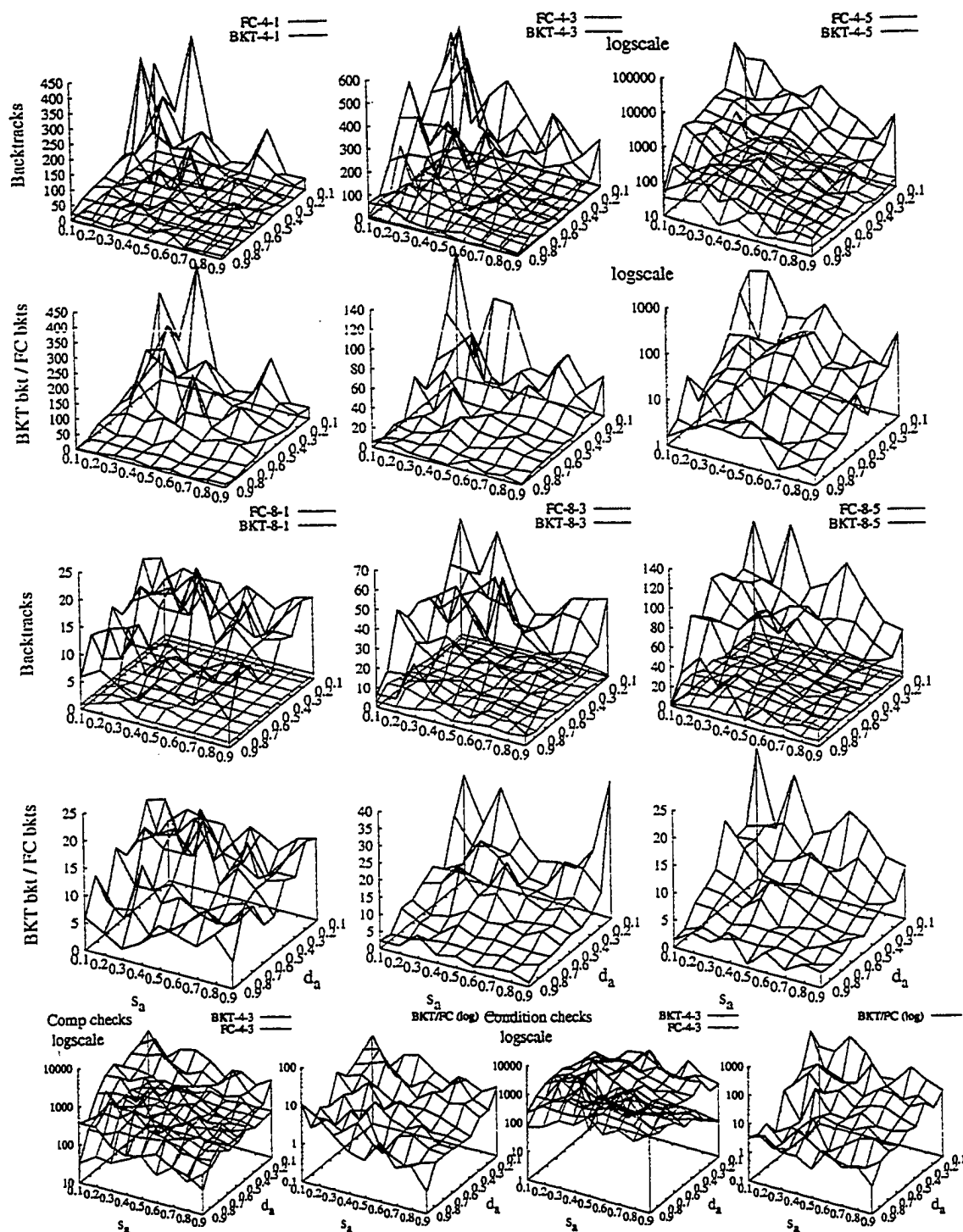


Figure 4-vi: Comparison between BT and FC using effort measures as in Figure 4-iv and Figure 4-v. Variation of effort with  $d_a$  and  $s_a$ . Fixed compatibility density of 0.4 (rows 1, 2, and 5) and 0.8 (rows 3 and 4).

#### Study 4: MAC and FC relative performance for finding all solutions

The test suite of this study consists of 60 problem classes of various value combinations for two parameters of the compatibility constraint graph: compatibility density,  $d_c$ , and compatibility satisfiability,  $s_c$ . In each  $(d_c, s_c)$  class, for each activity density,  $d_a$ , in the range  $[0.1 \dots 0.9]$  we graphed algorithm average behavior over 10 random problems as a function of activity satisfiability,  $s_a$ , in the range  $[0.1 \dots 0.9]$ . Algorithm performance was measured by counting six search operations: backtracking, checking activity conditions, including and excluding variables, and checking redundant and conflicting activations. Search cost for MAC was compared with search cost for FC across all six counters. The two algorithms search for all solutions. As expected, the larger the solution space, the greater the effort to compute them. Therefore, we also reported the average number of solutions for every 10 problems generated in a  $(d_c, s_c, d_a, s_a)$  topological point, and the variation of the number of solutions with  $s_a$ .

There were 6,480 graphs produced in this study that show the variation with  $s_a$  of a given effort counter (6 in total) for both FC and MAC on random problems in a given  $(d_c, s_c, d_a)$  class ( $60 \times 9 = 540$  classes in total). That is,  $6 \times 2 \times 540 = 6,480$ . We also plotted the variation of the number of solutions with  $s_a$  for all 540 random problem classes, and came up with a total of 7,020 graphs.

**Example 9.** An example of the results obtained in this study is shown in Figure 4-vii. The graphs plot the number of backtracks performed by FC and MAC, and the number of solutions found when solving random conditional CSPs generated with  $d_c=0.3$ ,  $s_c=0.3$ ,  $d_a=0.6$ , and with  $s_a$  varied in the range  $[0.1 \dots 0.9]$ . Two observations stand out: (1) MAC outperforms forward checking, and (2) the variation of algorithm effort (MAC and FC graphs, Figure 4-vii left) is similar to the variation of the number of solutions (Solutions graph, Figure 4-vii right) for the population of random problems in the given class.

△

The main objective of this study is to verify that the experimental evidence in Example 9

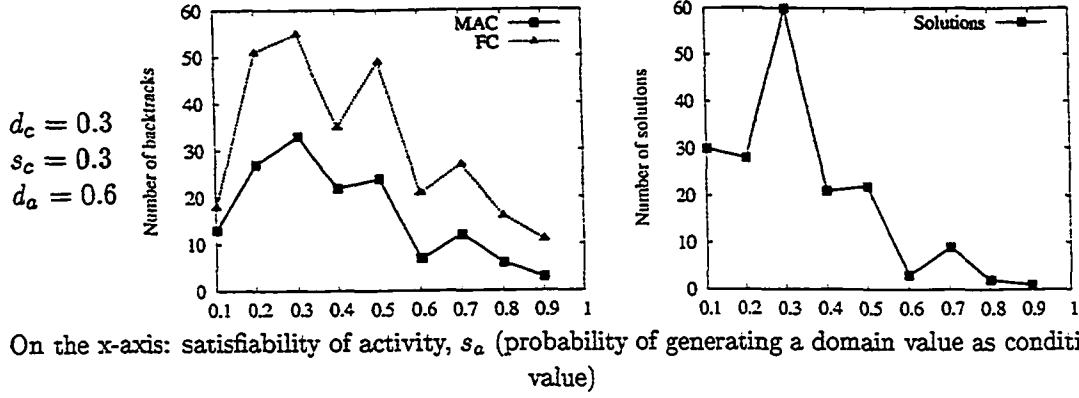


Figure 4-vii: Relative performance of FC and MAC measured by number of backtracks (left) and variation of number of solutions (right) as functions of  $s_a$ . Algorithms are run on random problems in  $(d_c, s_c, d_a) = (0.3, 0.3, 0.6)$  class. Performance and solution values are averages over sets of 10 problems for each  $(d_c, s_c, d_a, s_a)$  topological point.

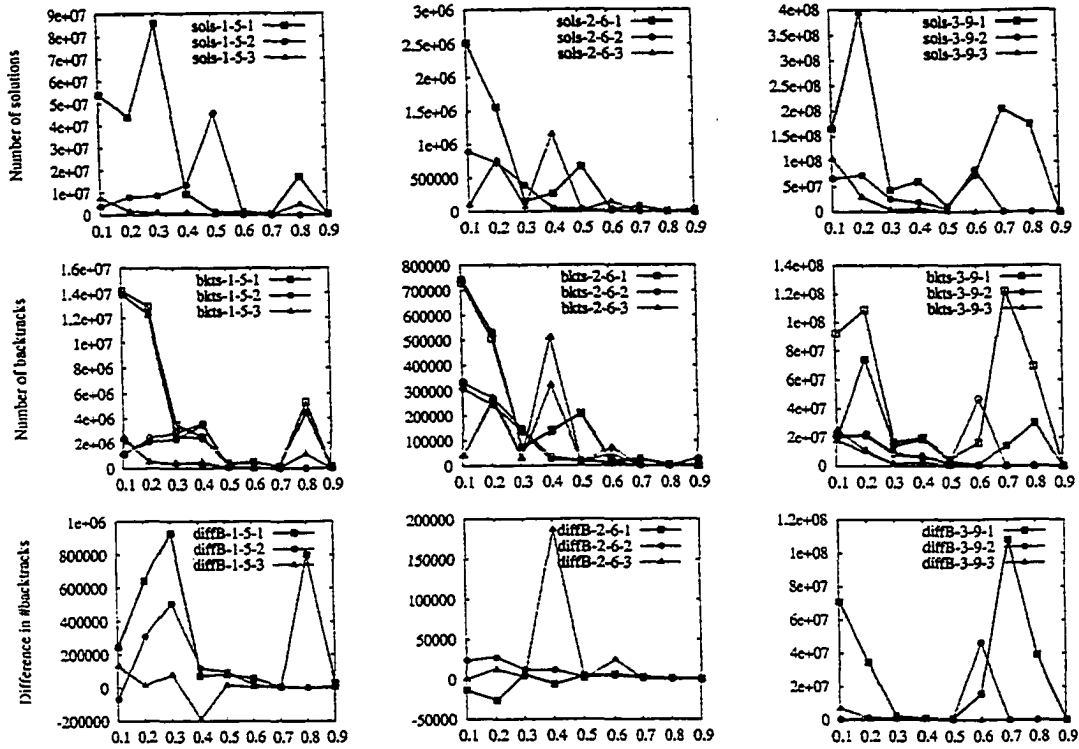
can be obtained across a more comprehensive test suite:

- On all counter measures of algorithm effort, MAC performs better than FC, and
- Search effort variation is significantly dependent on the size of the solution space.

The major empirical findings of this study are organized in four sub-studies that systematically cover all problem topologies and evaluate relative performance as follows:

- in terms of the number of backtracks on problems with
  - very large solutions sets, Study 2.1,
  - similarly small solution sets, Study 2.2,
  - extended topological coverage, Study 2.3,
- in terms of activity counters (condition checks, included and excluded variables, and redundant and conflicting activations) on very diverse problem populations, Study 2.4.

**Study 2.1. Problems with very large solution sets.** The largest number of solutions occurs for sparse compatibility and activity constraint graphs and high satisfiability of the compatibility constraints. Solving random problems with



On the x-axis: satisfiability of activity,  $s_a$  (probability of generating a domain value as condition value)

Figure 4-viii: For problems with very large solution sets (top), MAC and FC perform very similarly in terms of number of backtracks (middle). The test suite uses random problems in three  $(d_c, s_c)$  classes:  $(0.1, 0.5)$ ,  $(0.2, 0.6)$ , and  $(0.3, 0.9)$  for which  $d_a$  takes on low values of 0.1, 0.2, and 0.3.

- low density of both compatibility and activity constraint graphs, and
- higher satisfiability of compatibility constraints

shows that FC and MAC algorithm performance measured as number of backtracks is highly similar.

**Test Suite.** We compared the number of backtracks performed by MAC and FC when they run on problems in three  $(d_c, s_c)$  classes,  $(d_c = 0.1, s_c = 0.5)$ ,  $(d_c = 0.2, s_c = 0.6)$ , and  $(d_c = 0.3, s_c = 0.9)$ , of low compatibility density and relatively high compatibility satisfiability. In each class we restricted activity density  $d_a$  to low values of 0.1, 0.2, and 0.3.

**Results.** The results are plotted in Figure 4-viii. The top three graphs show the variation of the number of solutions of problems in each of the three classes. The notation sols-1-5-1, for example, is used to name a graph that plots the number of solutions of problems with compatibility density  $d_c = 0.1$ , compatibility satisfiability  $s_c = 0.5$ , and activity density  $d_a = 0.1$ . We observe that the lower  $d_a$ , the larger the solution space across the three classes, (0.1, 0.5), (0.2, 0.6), and (0.3, 0.8) classes.

The rest of the six graphs show how MAC and FC perform relative to each other. In the middle of the figure, we drew two graphs, MAC's number of backtracks and FC's number of backtracks, for each  $(d_c, s_c, d_a)$  problem class. Each graph pair is denoted by the same name, bkts -  $d_c - s_c - d_a$  and plotted with the same line style, corresponding to the line style we used for sols -  $d_c - s_c - d_a$  analogs. We observe that MAC and FC graphs of the number of backtracks follow closely the variation of their analogs that plot the number of solutions. To see how much better MAC does than FC, the bottom pictures have difference graphs, denoted by diffB, between FC's number of backtracks and MAC's number of backtracks. In general, MAC does fewer backtracks than FC, proportionally with the size of the solution space.

These results raise the question about MAC and FC relative performance when the solution space is significantly smaller. The answer to this question is the focus of the next experimental study.

**Study 2.2 Problems with similarly small solutions sets.** The objective is to delimit problem topologies of random conditional problems with much smaller solution sets of similar sizes. We want to evaluate how a scaled down solution space affects algorithm performance. We will examine how relative performance of MAC and FC algorithms changes with smaller solution sets, and whether MAC still outperforms FC.

**Test Suite.** The populations of problems that form our test suite have the density of the compatibility constraint graph,  $d_c$ , varied in steps of 0.1 in the range [0.3...0.6]. To control the number of solutions such that it does not exceed 100, we choose ranges of three compatibility satisfiability levels,  $s_c$ , specific to each of the four density values,  $d_c$ , and

obtained the following 12 problem sets:

$$\begin{array}{cccc}
 (d_c = 0.3, s_c = 0.2) & (d_c = 0.4, s_c = 0.3) & (d_c = 0.5, s_c = 0.4) & (d_c = 0.6, s_c = 0.5) \\
 (d_c = 0.3, s_c = 0.3) & (d_c = 0.4, s_c = 0.4) & (d_c = 0.5, s_c = 0.5) & (d_c = 0.6, s_c = 0.6) \\
 (d_c = 0.3, s_c = 0.4) & (d_c = 0.4, s_c = 0.5) & (d_c = 0.5, s_c = 0.6) & (d_c = 0.6, s_c = 0.7)
 \end{array}$$

Another control factor is the amount of activity these problems exhibit. We found that for constant density of activity  $d_a = 0.6$  and satisfiability of activity  $s_a$  in the high range of  $[0.5 \dots 0.9]$ , we obtain a suite of problem topologies that meet the requirement for similarly small solution sets.

**Results.** As shown at the top of Figure 4-ix, the graphs of the number of solutions for all  $(d_c, s_c)$  value pairs,  $\text{sols} - d_c - s_c$ , do not exceed the 100 mark and are mainly in the  $[0 \dots 20]$  range. On all these problems MAC does fewer backtracks than FC. In the four graphs in the middle of Figure 4-ix we have  $\text{MAC} - d_c - s_c$  and  $\text{FC} - d_c - s_c$  graphs that show how MAC and FC perform on all  $(d_c, s_c)$  problem classes in the test suite. At the bottom of the figure we draw  $\text{diffB} - d_c - s_c$  graphs that show the variation of the difference between the number of backtracks done by FC and MAC. The larger the number of solutions (that is, on problems with compatibility constraints of higher satisfiability  $s_c$ ), the larger the gain of MAC over FC. When we examine FC backtracks over MAC backtracks, we obtain the  $\text{FC/MAC} - d_c - s_c$  graphs (bottom row in Figure 4-ix). They show the factor by which MAC is more cost-effective than FC. Note that a subunit factor shows FC over performing when compared with MAC. The  $\text{FC/MAC} - d_c - s_c$  graphs illustrate that the cost saving improves overall by a factor of 2 up to 3 in the region of problems with larger solution sets. For these problems, the improvement factor tends to increase with values for  $s_a$  above 0.7 level. In the region of problems with fewer solutions, where  $s_c$  is set to the lowest levels for the four density groups, and again for high  $s_a$  values, MAC can be 4 to even 9 times better than FC. For these problems, however, the difference in number of backtracks is under 20.

**Study 2.3 Extending topological coverage.** Guided by the variation of the solution space with compatibility density in the previous two sub-studies, we isolated a density threshold of 0.3 that delimits very large solution sets from more manageable to very small sets in the presence of correlated compatibility satisfiability. The question that led to this

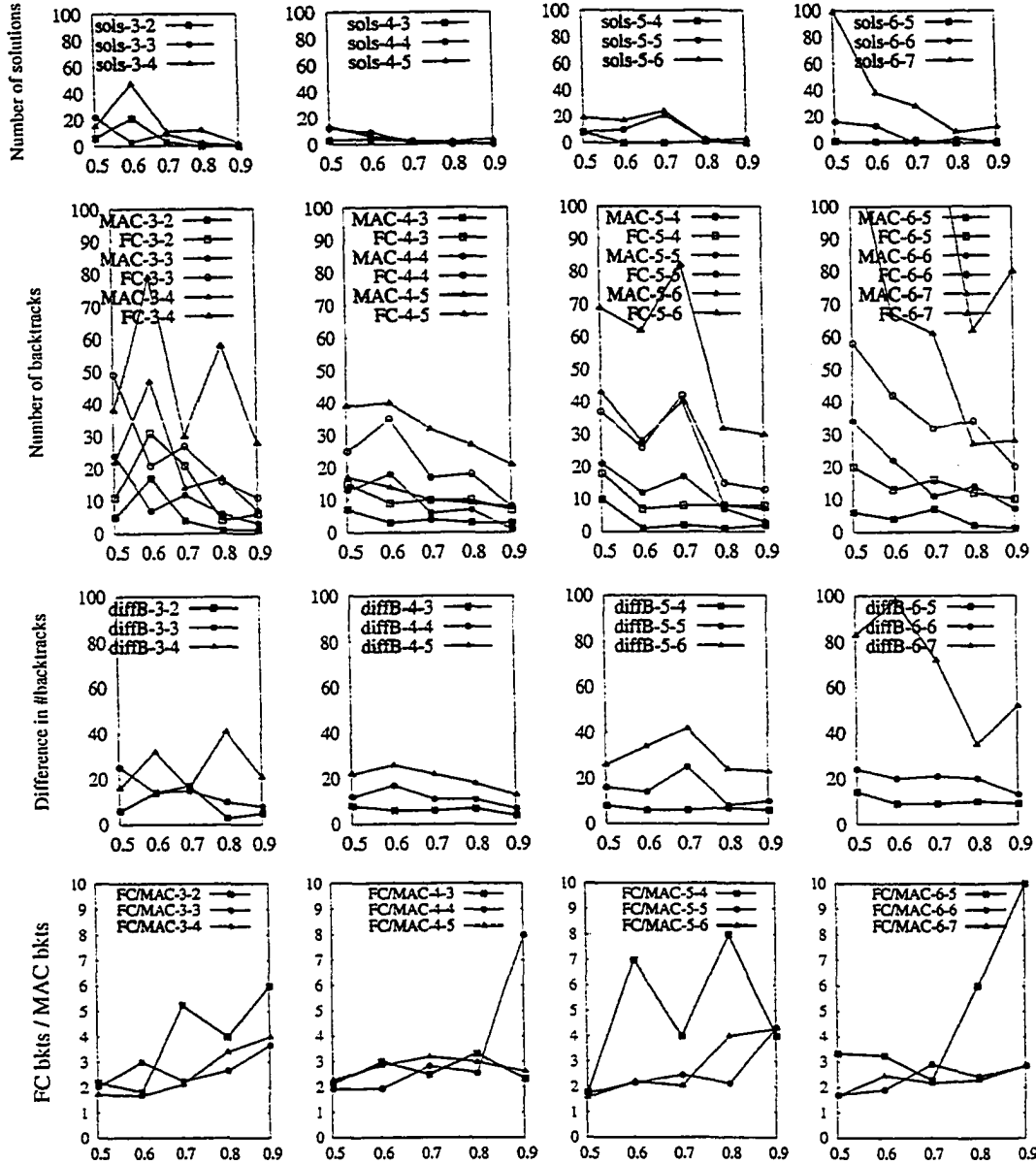


Figure 4-ix: Fixing  $d_a$  at 0.6 and varying  $s_a$  in the high range of  $[0.5 \dots 0.9]$ , we select  $(d_c, s_c)$  topologies for which the number of solutions is roughly the same (top). On all these problems MAC outperforms FC (rows 2 and 3). Cost effectiveness of MAC algorithm is measured by computing FC backtracks over MAC backtracks (bottom).

study is to find out how relative performance scales across a topological spectrum that is further extended but avoids very large solution sets.

Test Suite. Problem topologies in this study are defined by

- fixed activity density,  $d_a = 0.5$ ,
- three levels of compatibility satisfiability  $s_c$  of 0.5, 0.6, and 0.7,
- wider range of compatibility density  $d_c$  of 0.3 to 0.8,
- full range of activity satisfiability  $s_a$  of 0.1 to 0.9.

The variation of the number of solutions to problems in this topological spectrum is shown in Figure 4-x on a normal scale (top left) and log scale (bottom left). The graphs show that higher compatibility satisfiability  $s_c$  yields much larger solution sets for sparse compatibility constraint graphs (lower  $d_c$ ) and little domain activity (low  $s_a$ ). The three surfaces controlled by the three satisfiability values collapse to nearly zero for higher compatibility density  $d_c$  and activity satisfiability  $s_a$  equal to or greater than 0.6.

The data set in the upper right table in Figure 4-x that corresponds to the highest  $s_c = 0.7$  and full range of  $s_a$  draws attention to its lower right corner that has solution values of zero or close to zero. The same observation is shown in the second table below that has the same solution value distribution for all three satisfiability levels.

Results. Figure 4-xi (top) plots on a log scale the number of backtracks when MAC and FC run on the test suite problems. The graphs show that:

- MAC always outperforms FC.
- There is a strong dependency of algorithm effort on the size of the solution space:
  - the decline of the number of backtracks with larger compatibility density  $d_c$  and smaller activity satisfiability  $s_a$  is similar to the decline of the number of solutions with the same parameter variation, and
  - peak values of the number of solutions correlate with the corresponding peak values of the six effort surfaces (Table 4.1): with the exception of two entries,  $s_c = 0.6$  and  $s_c = 0.7$  for  $s_a = 0.1$ , the number of solutions and number of backtracks performed by MAC and FC have the same order of magnitude.

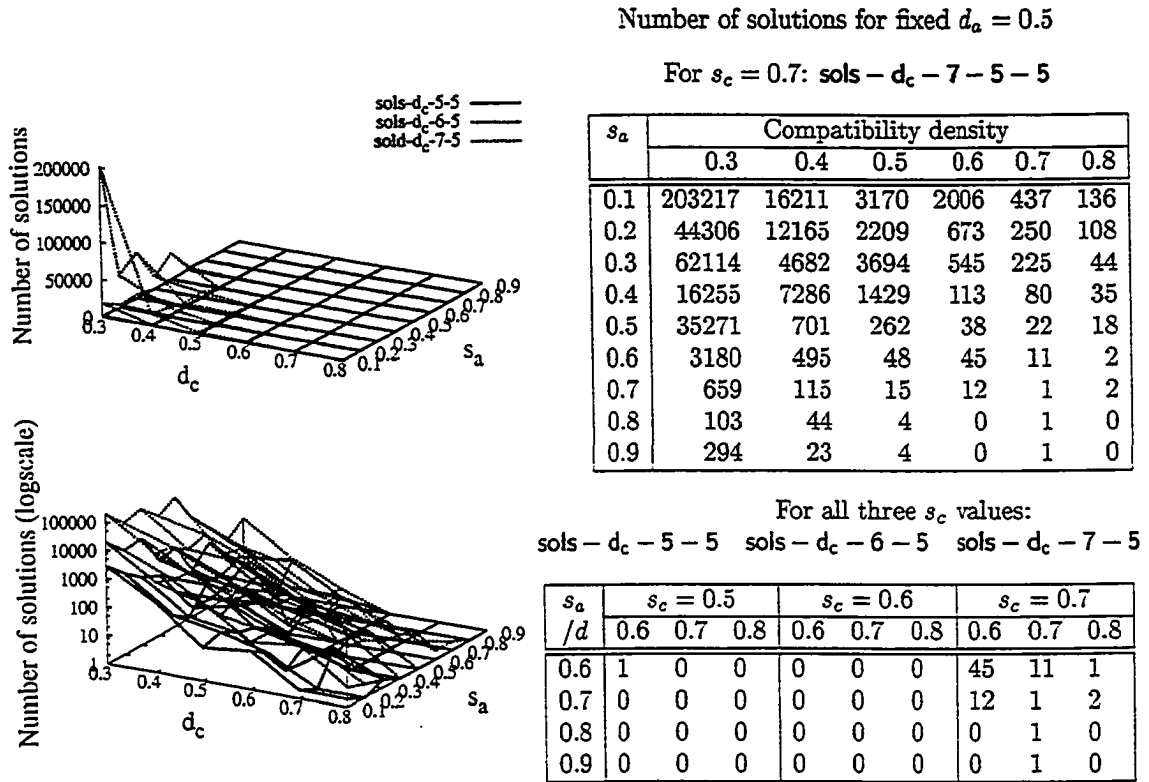


Figure 4-x: (Left) Variation of number of solutions for fixed  $d_a = 0.5$ , three satisfiability levels of 0.5, 0.6, and 0.7, and variable  $d_c$  in  $[0.3 \dots 0.8]$  range and  $s_a$  in  $[0.1 \dots 0.9]$  range: linear scale (top) and log scale (bottom). (Right) Data sets for the number of solutions plotted by sols -  $d_c - 7 - 5$  (top) and data sets over ranges of high  $d_c$  in  $[0.6 \dots 0.8]$ , and high  $s_a$  in  $[0.6 \dots 0.9]$ , for the number of solutions of all graphs categories (bottom).

Table 4.1: For  $s_c$  in  $[0.5 \dots 0.7]$ , low  $s_a$  in the range  $[0.1 \dots 0.3]$  and low  $d_c$  of 0.3, the number of solutions has values of an order of magnitude equal to or greater only by one than the number of backtracks performed by FC and MAC algorithms.

$d_c = 0.3$	$s_c = 0.5$			$s_c = 0.6$			$s_c = 0.7$		
$s_a$	sols	#bkts		sols	#bkts		sols	#bkts	
		FC	MAC		FC	MAC		FC	MAC
0.1	2902	1649	1541	19785	7887	7379	203217	85601	81743
0.2	734	717	480	4721	2627	2266	44306	25865	22239
0.3	245	273	183	5753	4482	3560	62114	30112	26750

Figure 4-xi (middle) plots the difference in number of backtracks performed by the two algorithms. Peak difference values increase from 250 to 1000 to 4500 with compatibility satisfiability  $s_c$  on problem populations of low compatibility density,  $d_c$ , and low activity

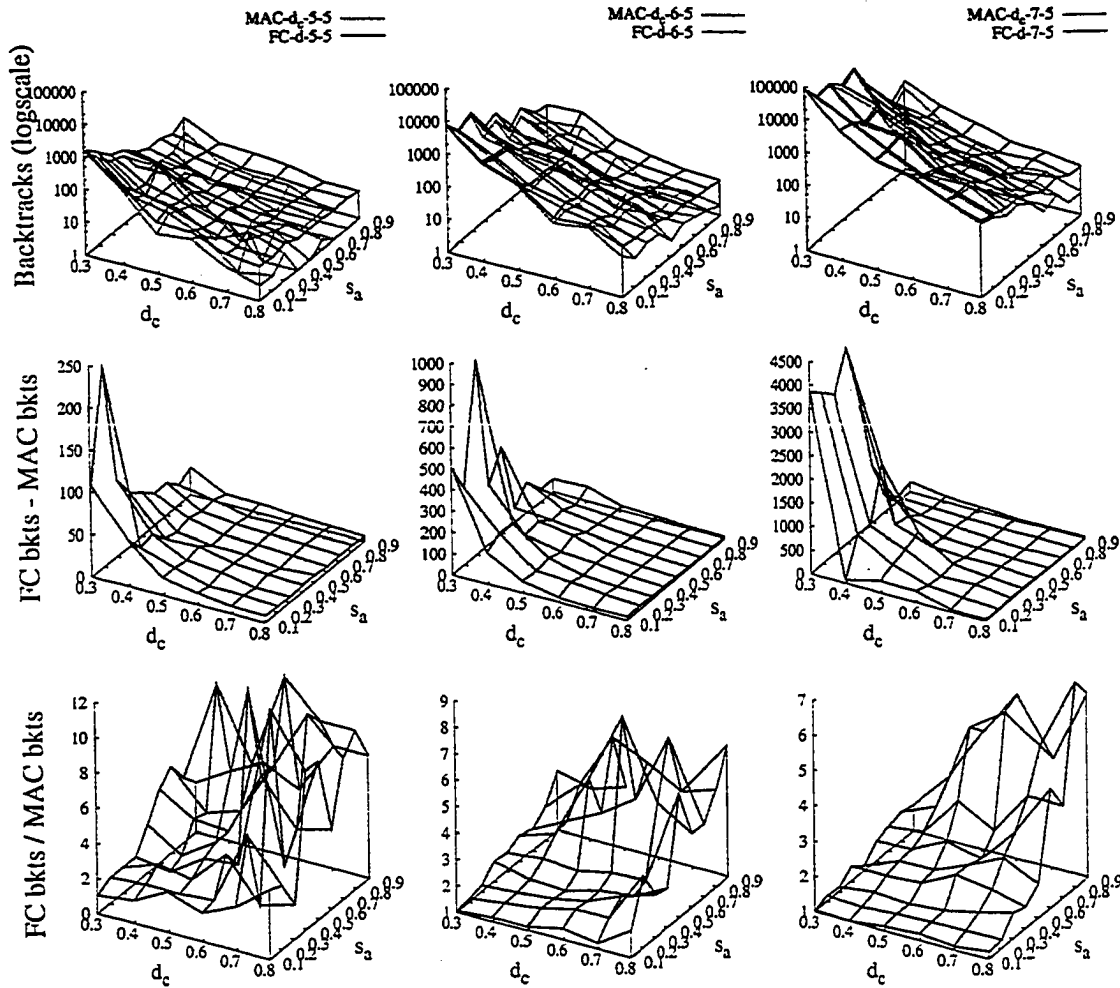


Figure 4-xi: Variation of number of backtracks on the problem set in Figure 4-x, with each column corresponding to different satisfiability value: 0.5 (left), 0.6 (middle), and 0.7 (right). Comparison between MAC and FC performance using a log scale (top); difference in number of backtracks between FC and MAC (middle); performance factor of FC backtracks over MAC backtracks (bottom).

satisfiability,  $s_a$ . As the  $d_c$  and  $s_a$  parameters increase, backtrack difference values decrease considerably for all compatibility satisfiability  $s_c$  levels. A way of “zooming in” on algorithm performance for problem populations delimited by  $d_c = -s_a$  and maximum values of  $d_c = 0.8$  and  $s_a = 0.9$  (upper triangle above second diagonal of the  $(d_c, s_a)$  plane), is to look at backtrack ratio variation in that region (Figure 4-xi bottom). These cost effectiveness surfaces show that MAC is better than FC by a factor no greater than 2 on problems for

which we reported the largest backtrack differences, and by a factor of 5 to 10 on problems for which backtrack differences reach a very low plateau. The smaller the difference in the number of backtracks, the more rapidly the maximum gain is reached. However, MAC's best performance is observed only on problems with very few or no solutions, whose solving requires the fewest backtracks, regardless of the solving method. The data sets for problems with  $d_c$  and  $s_a$  larger than 0.6, given in Table 4.2, list the average number of solutions,  $s$ , of FC backtracks,  $f$ , and MAC backtracks  $m$ , for each problem class. The backtracks values are small and, even if MAC is many times better than FC in this problem region, that happens on very easy problems.

Table 4.2: For  $d_c$  and  $s_a$  larger than 0.6, problems have very few or no solutions, and MAC and FC perform very few backtracks.

$s_a$ / $d_c$	$s_c = 0.5$									$s_c = 0.6$									$s_c = 0.7$								
	0.6			0.7			0.8			0.6			0.7			0.8			0.6			0.7			0.8		
	s	f	m	s	f	m	s	f	m	s	f	m	s	f	m	s	f	m	s	f	m	s	f	m	s	f	m
0.6	1	13	2	0	12	3	0	9	2	13	30	4	0	23	5	0	19	8	45	164	87	11	87	30	2	48	11
0.7	0	13	2	0	10	0	0	8	0	0	24	6	0	20	3	0	15	8	12	90	38	1	47	13	2	61	17
0.8	0	11	1	0	9	1	0	9	1	3	20	4	0	17	4	0	14	6	0	49	9	1	57	12	0	34	5
0.9	0	12	3	0	7	0	0	7	1	1	18	2	0	13	3	0	12	8	0	51	9	1	45	11	0	31	5

**Study 2.4 Activity Effort.** We conclude the result analysis of the experiments in Study 2 with probing counters that measure algorithm activity effort. These counters gauge algorithm effort of dynamically changing the initial problem as dictated by enforcing activity constraints. Algorithm activity effort associated with activity constraints is measured by counting:

- condition checks performed with the instantiation of condition variables,
- the number of variables successfully included or excluded as the result of satisfying activity constraints,
- redundant activations, which unnecessarily reset variable's activity status to values

that have been already set,

- conflicting activations, which invalidate variables activity status.

The objective is to:

- observe the correlation between solution set size and algorithm activity effort, and
- compare MAC and FC performance in terms of activity counters.

Test Suite. The variation of activity counters was examined in two settings:

- the control parameters of activity, density and satisfiability, are varied in the  $[0.1 \dots 0.9]$  range, while their standard counterparts are fixed:  $d_c = 0.3$  and  $s_c = 0.6$ ;
- the density of both compatibility and activity constraints are varied,  $d_c$  in  $[0.3 \dots 0.8]$  and  $d_a$  in  $[0.1 \dots 0.9]$ , while satisfiability parameters are fixed:  $s_c = 0.6$  and  $s_a = 0.4$ .

Results. The major findings of this experiments are:

- the largest solution sets occur in problem regions of:
  - lower conditionality,  $d_a$  and  $s_a$  in the first half of their variation interval  $[0.1 \dots 0.5]$ ,
  - opposite ranges of variability for density of both types of constraints: low compatibility density  $c_c$  of 0.3 and 0.4, as opposed to high activity density  $d_a$  in  $[0.5 \dots 0.9]$ .
- MAC counts less effort than FC on all activity tasks except for conflicting activations.

Experimental results for problems of variable conditionality are plotted in Figure 4-xii and Figure 4-xiii. Experimental results for problems of variable density are plotted in Figure 4-xiv and Figure 4-xv. In both settings we show on a normal scale (left columns) and logarithmic scale (middle columns) the variation of the number of solutions and activity counters. The degree to which one algorithm outperforms the other is reported by the variation of the ratio between MAC and FC corresponding counters (right columns). In

two instances we use the variation of difference between MAC and FC measured effort (Figure 4-xiii middle column).

We observe that experiments in which conditionality parameters, density and satisfiability of activity constraints,  $d_a$  and  $s_a$ , were varied within the full range, the solution sets are more than twice as large in comparison with solutions to problems where activity satisfiability,  $s_a$ , was fixed at 0.4 and both density parameters,  $d_c$  and  $d_a$ , were varied. Consequently, in the first set of experiments MAC and FC performed more activity operations than in the second set. On average, MAC reduced the number of included variables by a factor of 2 and the number of excluded variables by a factor of 3. Overall, MAC did half as many condition checks as FC on problems of variable conditionality,  $d_a$  and  $s_a$ . Similar improvement was maintained on problems of variable density with the exception of high the density region.

In both types of experiments, from all activity effort counters MAC recorded the largest improvement over FC for processing fewer redundant activations, by an average factor of 5. The activity task that was more costly for MAC was processing conflicting activations. This is caused by MAC's specialized activity arc-consistency, which finds activity constraints that invalidate the activity status of future variables. Condition values of those conflicting activity constraints are pruned from the domains of future variables whose activity status is invalidated. By processing more conflicting activations, MAC checks fewer condition values and, consequently, does fewer activity constraint checks.

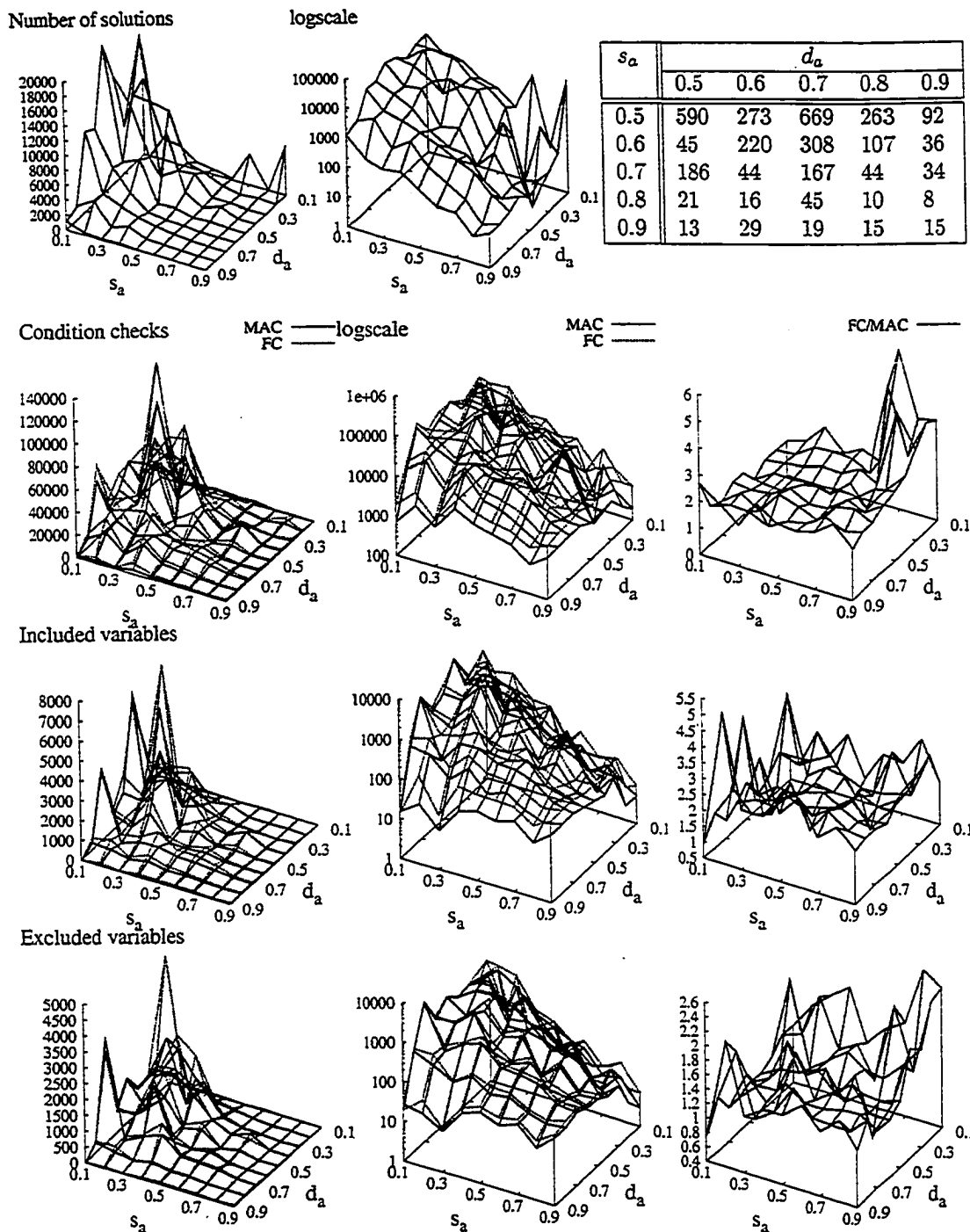


Figure 4-xii: Problems of variable conditionality:  $d_a$  and  $s_a$  varied in  $[0.1 \dots 0.9]$  and fixed  $d_c = 0.3$  and  $s_c = 0.6$ . (First row) Number of solutions on normal scale (left) and logarithmic scale (middle); significantly smaller solution sets for  $d_a$  and  $s_a$  in the second half of their interval (right). Variation of activity counters: condition checks (row 2), included variables (row 3), and excluded variables (row 4). Relative performance of FC and MAC is shown for each activity counter: both MAC and FC effort surfaces on normal scale (left) and logarithmic scale (middle), and ratio of MAC and FC corresponding counters (right).

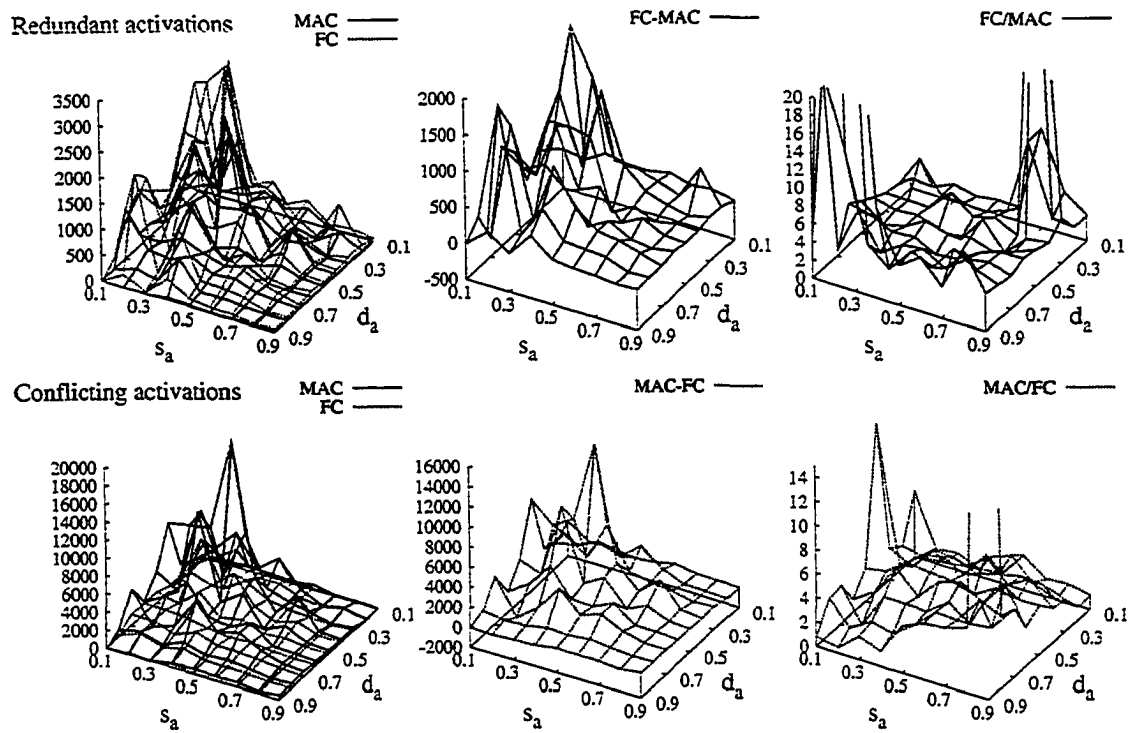


Figure 4-xiii: Continuation of Figure 4-xii. Activity counters: redundant activations (top) and conflicting activations (bottom). MAC checks fewer redundant activity constraints, but more conflicting activity constraints.

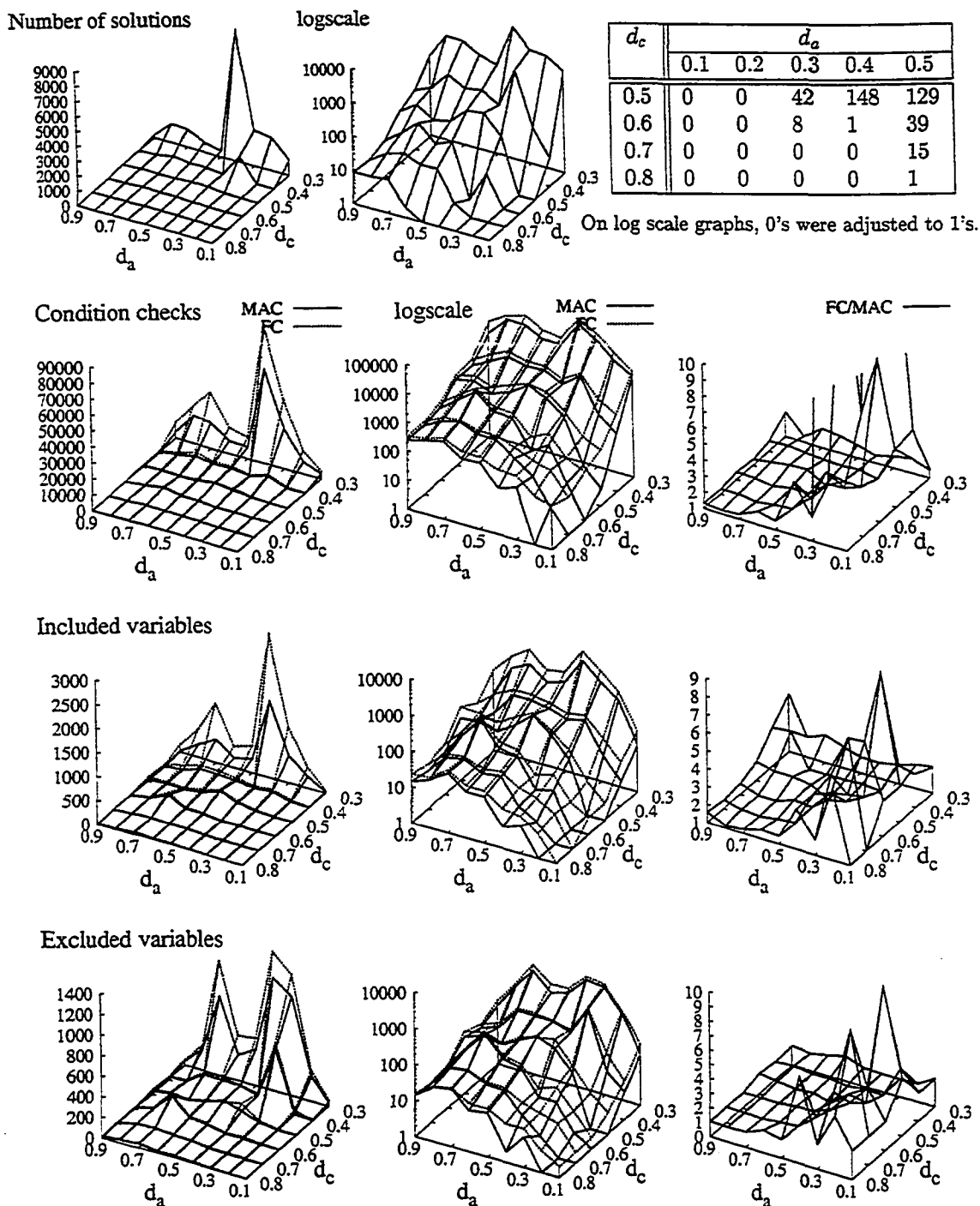


Figure 4-xiv: Problems of variable densities,  $d_c$  and  $d_a$ , and fixed satisfiability parameters,  $s_a = 0.4$  and  $s_c = 0.6$ . (First row) Number of solutions on normal scale (left) and logarithmic scale (middle); largest solution sets controlled by low compatibility density,  $d_c$ , of 0.3 and 0.4. Variation of activity counters and relative performance of FC and MAC are reported in the same fashion as in previous experiment in Figure 4-xii.

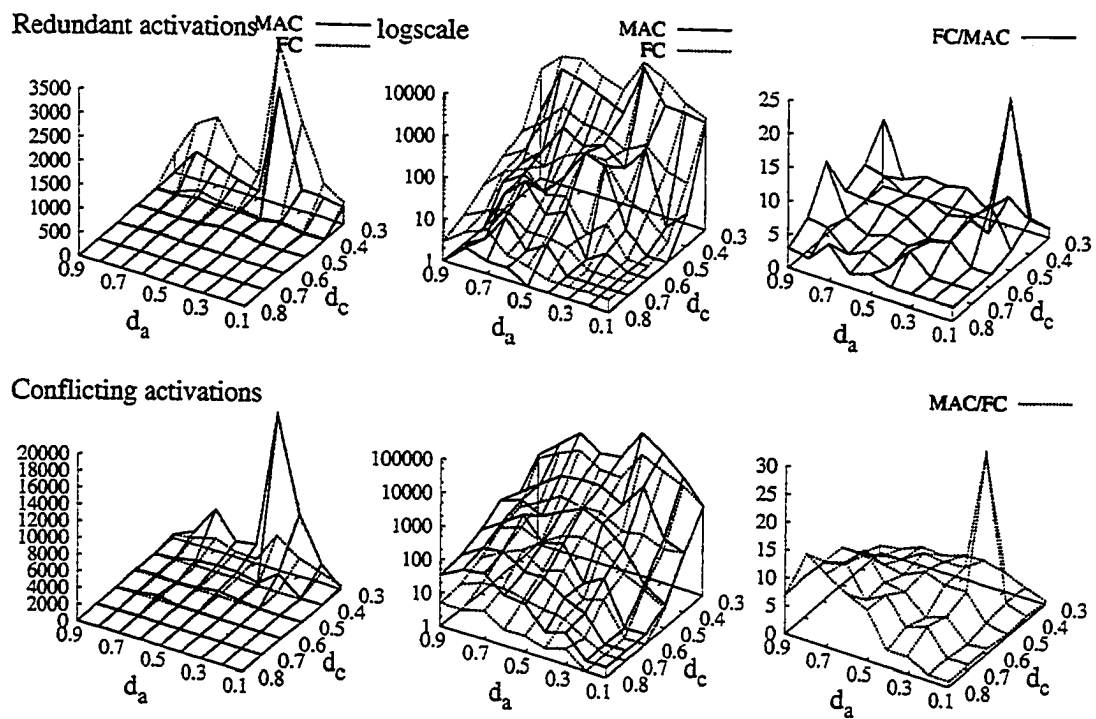


Figure 4-xv: Continuation of Figure 4-xiv. Variation of redundant (top) and conflicting (bottom) activations. MAC checks fewer redundant activity constraints and more conflicting activity constraints .

## 4.4 Summary

In this chapter we considered the practical approach of benchmarking solving methods by using random conditional CSPs. The algorithms developed in the previous chapter were tested in experiments covering diverse populations of randomly generated problems.

Like random standard CSPs, random conditional CSPs are characterized by problem size as well as density and satisfiability of compatibility constraints. Problem activity introduces new parameters with which Richard Wallace extended his random standard CSP generator (Wallace 1996) to produce random conditional CSPs.

We systematically varied the generator parameters to generate large problem sets with diverse topologies. From these problems we designed test suite on which we analyzed algorithm relative performance with regard to execution time and counters associated with representative search operations.

Experimental findings show that

- Forward checking always wins over backtrack search and, in terms of execution time, the gain is one to two orders of magnitude.
- Maintaining arc consistency significantly outperforms forward checking on hard problems.
- All three algorithms correctly produce the same number of solutions when run on identical problem instances to find all solutions. These results provide empirical evidence with regard to the algorithms' correctness.

## CHAPTER 5

### ON REFORMULATING CONDITIONAL CSPS

Conditional CSPs add a special type of constraint, called an activity constraint, to standard CSP. The purpose of these constraints is to condition which variable sets participate in final solutions. These variables are called active variables. The representational means of problem activity in a conditional CSP are intuitive and easy to use for modeling conditional selection of active variables. The focus of this chapter is to examine how conditional CSP behavior can be reformulated using traditional components of standard CSPs: variables, values, and regular constraints (what we call compatibility constraints in the conditional CSP model). The motivation for moving the problem representation from conditional constraint satisfaction to the standard domain is given by the prominence and maturity of the constraint satisfaction classical paradigm and the effectiveness of its solving methods. The issues we address in this chapter are:

- the feasibility of transforming a conditional CSP into a standard CSP: what are the challenges and how they can be overcome, and
- an experimental comparison between solving a reformulated standard CSP using classical algorithms and directly solving the original conditional CSP using the new algorithms developed in the previous chapter.

#### 5.1 Introduction

The reformulation of a conditional CSP into an equivalent standard CSP was first reported by Mittal and Falkenhaimer, although they do not describe how exactly the transformation is done. They consider the addition of a special value, called “null”, to the domains of

non-initial variables. A variable instantiation with a “null” value indicates that the variable is not used in problem solutions. Mittal and Falkenhainer mention that “appropriate transformations of all constraints” have to be made to take into account the new null value such that the transformed constraints be “trivially satisfied” for value combinations that include null values. No specific descriptions are given about these transformations. Mittal and Falkenhainer report that the comparison between solving a conditional CSP directly and solving a null-based reformulation on a set of examples shows significant gains in all performance metrics for the direct method<sup>1</sup>. However, no description of the direct method is given.

Mittal and Falkenhainer’s formalization of dynamic constraint satisfaction is reviewed in (Haselböck 1993), where the definitions of dynamic constraint network, consistency, solution, and irreducible (or minimal) solutions are restated using a slightly different theoretical formalism. Relevant to reformulation, Haselböck claims that exclusion activity constraints are not really necessary, and demonstrates how they can be expressed as conventional compatibility constraints. However, a complete reformulation of conditional CSP into standard CSP is briefly qualified as “not very straightforward” and inefficient, since the addition of “dummy domain values (like inactive) for all possibly unused variables” increases the size of the problem and, consequently, the search effort.

The feasibility of obtaining a null-based CSP formulation from a conditional CSP is first examined in-depth in (Gelle 1998). Gelle proceeds with proposing first an algorithm (Algorithm  $A_1$ , page 111) that automates this transformation and produces a solution set that contains supersets of the solution assignments obtained in the original problem. A superset assignment satisfies the compatibility constraints. However, it adds to the original corresponding solution, let us call it  $s$ , values that instantiate variables which are not included in the original problem by  $s$ . The reformulation algorithm uses a preliminary

---

<sup>1</sup>The direct method implements a subset of the conditional CSP language (without activity constraints of exclusion) by “extending a conventional backtrack search CSP” with “forward checking to propagate all compatibility constraints” (Mittal & Davis 1989)

transformation of the exclusion activity constraints into compatibility constraints according to Haselböck's procedure. Compatibility and inclusion activity constraints are then transformed into regular constraints that handle null values.

Gelle modifies the algorithm to find the exact solution set under the assumption that each variable is activated by at most one activity constraint (Algorithm  $A_2$ , page 111). If this assumption is relaxed, Gelle proposes and shows with an example that activity constraints that trigger the same activation can be collapsed into one activity constraint, and then the resulting activity constraint can be transformed into an equivalent compatibility constraint. The drawback of this transformation, Gelle observes, is that it violates the locality of change in the reformulated CSP when a local change, such as the addition of an activity constraint, occurs in the original problem.

We will show in the next section that the idea of clustering multiple activations into one and then transforming it as a single activation can be used (1) to formalize the transformation, and (2) to design a reformulation algorithm for a restricted class of conditional CSPs. A problem arises if cluster activations of the same variable form cycles. The cluster activation transformation we propose assumes that problems do not have activity cycles. Next, under the same assumption, we derive an algorithm that allows for preserving locality of transformation when the original problem changes locally. At the end of the section we present two examples to illustrate the issues introduced by activity cycles, and propose a solution.

In Section 5.3, the implementation of the algorithm is evaluated on CSP reformulations of random conditional CSPs and its performance is compared with the direct method, *CCSP\_SolveMac*, applied to the original conditional CSPs. The chapter concludes with a summary section.

## 5.2 Reformulation Algorithms

A high-level description of the reformulation of a conditional CSP into a null-based standard CSP has three generic components:

- Reformulation of value domains. Null values,  $N$ , are added to the domains of all variables which are not initial variables.
- Reformulation of compatibility constraints. Compatibility constraints are expanded with all possible tuples that contain at least one  $N$  value.
- Reformulation of activity constraints. Inclusion and exclusion activity constraints are transformed into conventional constraints.

These components structure the design of a generic reformulation algorithm, *NullBasedReformulate* (see Algorithm 5.1), which transforms a conditional CSP,  $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ , into a null-based equivalent CSP,  $\mathcal{P}_R = \langle \mathcal{V}_R, \mathcal{D}_R, \mathcal{C}_R \rangle$ .

**Algorithm 5.1.** *Null-based CSP reformulation,  $\mathcal{P}_R = \langle \mathcal{V}_R, \mathcal{D}_R, \mathcal{C}_R \rangle$ , of a conditional CSP,  $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ .*

```

NullBasedReformulate( $\mathcal{P}, \mathcal{P}_R$ ) {
   $\mathcal{V}_R \leftarrow \mathcal{V}$ 
  refDomains( $\mathcal{V}, \mathcal{V}_I, \mathcal{D}, \mathcal{D}_R$ )
  refCompatibility( $\mathcal{C}_C, \mathcal{V}, \mathcal{D}_R, \mathcal{C}_R$ )
  refInclusion( $\mathcal{C}_A, \mathcal{V}, \mathcal{D}_R, \mathcal{C}_R$ )
  refExclusion( $\mathcal{C}_A, \mathcal{V}, \mathcal{D}_R, \mathcal{C}_R$ )
} // end NullBasedReformulate()

```

The reformulation leaves the set of variables unchanged, that is,  $\mathcal{V}_R = \mathcal{V}$ . Reformulations of domains, compatibility constraints, and inclusion and exclusion activity constraints are delegated to four procedures: *refDomains*, *refCompatibility*, *refInclusion*, and *refExclusion*.

In the rest of this section all the reformulation procedures are exemplified on the sample problem (or slight variations of it) given in the following example.

**Example 10.** The simple conditional CSP problem,  $\mathcal{P}_1$  (Figure 5-i), is derived from an example originally given in (Mittal & Falkenhainer 1990) and used in (Gelle 1998).

Reformulation of variable domains is immediate. Compatibility constraint reformulation is straightforward too. To produce equivalent, ordinary constraints we add to the original allowed tuples new tuples to satisfy the constraint when at least one variable is not active

$$\begin{aligned}
\mathcal{P}_1 &= \langle \mathcal{V}_1, \mathcal{D}_1, \mathcal{V}_{I1}, \mathcal{C}_{C1}, \mathcal{C}_{A1} \rangle \\
\mathcal{V}_1 &= \{v_1, v_2, v_3\} \\
\mathcal{D}_1 &= \{D_1, D_2, D_3\} \\
&\quad D_1 = \{a, b\}, D_2 = \{c, d\}, D_3 = \{e, f\} \\
\mathcal{V}_{I1} &= \{v_1, v_2\} \\
\mathcal{C}_{C1} &= \{c_1, c_2\} \\
&\quad c_1 = C(v_1, v_2) = \{(ad) (bc)\} \\
&\quad c_2 = C(v_1, v_2, v_3) = \{(bce) (bcf) (acf) (bde) (ade) (bdf) (adf)\} \\
\mathcal{C}_{A1} &= \{a_1\} \\
&\quad a_1 = A(v_1, v_3) : v_1 = b \xrightarrow{\text{incl}} v_3 \\
\text{sol}(\mathcal{P}_1) &= \{\{v_1 = a, v_2 = d\}, \{v_1 = b, v_2 = c, v_3 = e\}, \{v_1 = b, v_2 = c, v_3 = f\}\}
\end{aligned}$$

Figure 5-i: Simple conditional CSP example,  $\mathcal{P}_1$ 

△

(undefined or excluded). The principle of this transformation originates from (Mittal & Falkenhainer 1990) and was first applied by Gelle in the transformation she proposes in Algorithm  $A_1$ , (Gelle 1998). This transformation is also used to indirectly transform exclusion activity constraints, which are first rewritten as compatibility constraints (Haselböck 1993).

Difficulties arise with transforming inclusion activity constraints. Gelle's transformation of inclusion activity constraints in Algorithm  $A_1$  generates extraneous solutions that are not produced by the original problem. To eliminate them, the transformation is refined in Algorithm  $A_2$ , which requires that peer exclusion activity constraints be added to the original problem. These constraints have to be reformulated as compatibility constraints prior to conditional-to-standard transformation. Algorithm  $A_2$  is correct, Gelle argues, as long as non-initial variables are made active by *single activations*.

The case of multiple activity constraints that independently activate the same variable, which we call *cluster activations*, needs a different treatment. Gelle proposes that cluster activations be first collapsed into one inclusion activity constraint and then be transformed into a regular constraint. Although Gelle does not formalize the transformation, she gives an example to show that reformulating these activations individually by applying Algorithm  $A_2$ , leads to a too restrictive constraint and an incomplete algorithm.

There is an inherent disadvantage to the transformation of cluster activations by synthesizing a singular inclusion activity constraint prior to the actual reformulation. Gelle points out that this transformation imposes the restriction that all inclusion activity constraints be known before hand. Consequently, the transformation does not allow for incremental introduction of additional constraints.

Prompted by the reformulation challenges exposed in Gelle’s feasibility study, we develop a framework in which we:

- Streamline the reformulation of conditional CSPs that are restricted to *single activations*, where active variables originate from single activations. We implement this transformation by the *refSingleInclusion* algorithm.
- Introduce a formalism for transforming *cluster activations*, where active variables are possibly targeted by a cluster of inclusion activity constraints. We implement this formalism by the *refClusterInclusion* algorithm.
- Derive an incremental version for reformulating cluster activations that preserves locality of change, and implement the *refIncrementalInclusion* algorithm.
- Present the activity cycle problem and solve it.

The section is organized around the contributions listed above. We start with conditional CSPs that have only *single activations* and present procedures for transforming all problem components: domains, compatibility constraints, and activity constraints. We relax the assumption about variable activation and allow *cluster activations*. Thus, we extend the framework with a new algorithm for transforming cluster activations. This algorithm is then improved to process the original problem in an incremental fashion. For these transformations we have enforced the assumption that activations do not exhibit the activity cycle problem. We conclude with a description of the activity cycle problem, and a more general reformulation algorithm that solves it.

### 5.2.1 Single Activations

#### Reformulation of Value Domains

Assigning value  $N$  to a variable  $v$  in a solution,  $s_R$ , of a null-based CSP reformulation means that  $v$  does not participate in the corresponding solution,  $s$ , in the original conditional CSP.  $v$ 's non-participation in solution  $s$  is shown by  $v$ 's activity status as undefined or excluded, which occurs in one of the following cases:

1. None of the activity constraints that activates  $v$  is satisfied by  $s$ , or
2. There is at least one exclusion activation whose condition is satisfied by  $s$ .

The transformation of value domains is simple. Algorithm 5.2 describes the *refDomains* procedure which produces a reformulated domain set,  $\mathcal{D}_R$ , from the original domain set,  $\mathcal{D}$ , of all variables,  $\mathcal{V}$ . First,  $\mathcal{D}_R$  is initialized with the domains of the initial variables. Then variables which do not belong to the initial variable set,  $\mathcal{V}_I$ , have their domains extended with a new value, called null and denoted by  $N$ .

**Algorithm 5.2.**  *$\mathcal{D}_R$  reformulation of the domains of values  $\mathcal{D}$  in a conditional CSP with  $\mathcal{V}$  variables and  $\mathcal{V}_I$  initial variables.*

```

refDomains( $\mathcal{V}, \mathcal{V}_I, \mathcal{D}, \mathcal{D}_R$ ) {
   $\mathcal{D}_R \leftarrow$  domains of  $\mathcal{V}_I$  variables
  for each ( $v \in \mathcal{V} - \mathcal{V}_I$ ) {
     $D_{vR} \leftarrow D_v \cup \{N\}$ 
     $\mathcal{D}_R \leftarrow \mathcal{D}_R \cup D_{vR}$ 
  }
} // end refDomains( )

```

#### Reformulation of Compatibility Constraints

How do null-extended variable domains affect the transformation of the compatibility constraints? The idea behind this transformation is that compatibility constraints,  $\mathcal{C}_C$ , are trivially satisfied if some of their variables are not active: their activity status is either undefined or excluded. In the allowed tuples of the corresponding reformulated constraints,  $\mathcal{C}_R$ , participation of constraint variables as undefined or excluded is indicated by the value

$N$ . Adding  $N$  values for some of the variables on which reformulated constraints are defined must leave unchanged the set of disallowed tuples of those constraints.

Let us denote the set of disallowed tuples of some constraint  $c$ , defined on variables  $var(c) = v_{c_1}, \dots, v_{c_k}$ , by  $\bar{c}$ , which is the complement of  $c$  with regard to the cross-product of the domains of the constraint variables,  $D_{v_{c_1}}, \dots, D_{v_{c_k}}$ . That is,  $\bar{c} = (D_{v_{c_1}} \times \dots \times D_{v_{c_k}}) - c$ . The exact value combinations which are not allowed in the compatibility constraint  $c$  remain disallowed in the corresponding reformulated constraint  $c_R$ . This means that, when expressing  $c_R$  by enumerating its allowed value combinations, we have to add all tuples that have at least one  $N$  value. The set of additional tuples is computed from the Cartesian product of the reformulated variable domains,  $D_{v_{c_i}R}$ , from which we exclude the disallowed tuples of the original constraint,  $\bar{c}$ .

Algorithm 5.3 describes the *refCompatibility* procedure that transforms compatibility constraints  $C_C$  in a conditional CSP  $\mathcal{P}$  into equivalent constraints  $C_R$  in  $\mathcal{P}_R$ , a null-based reformulation of  $\mathcal{P}$ .

**Algorithm 5.3.**  $C_R$  reformulation of compatibility constraints  $C_C$ .

```

refCompatibility( $C_C, \mathcal{V}, \mathcal{D}_R, C_R$ ) {
   $C_R \leftarrow \emptyset$ 
  for each ( $c \in C_C$ ) {
    let  $\{v_{c_1}, \dots, v_{c_k}\}$  be  $var(c)$ 
     $\bar{c} \leftarrow D_{v_{c_1}} \times \dots \times D_{v_{c_k}} - c$ 
     $c_R \leftarrow D_{v_{c_1}R} \times \dots \times D_{v_{c_k}R} - \bar{c}$ 
     $C_R \leftarrow C_R \cup \{c_R\}$ 
  } // end for
} // end refCompatibility()

```

The two procedures for transforming domains and compatibility constraints are exemplified as follows.

**Example 11.** The transformation of variable domains in Example 10 affects only non-initial variables and, consequently, constraints involving non-initial variables. Thus, only one variable,  $v_3$ , changes its domain with the addition of the value  $N$ . The domains of initial variables  $v_1$  and  $v_2$  remain unchanged:

$$D_{1R} = D_1, D_{2R} = D_2, D_{3R} = D_3 \cup \{N\} = \{e, f, N\}$$

The compatibility constraint  $c_1 = C(v_1, v_2)$  does not change, thus  $c_{1R} = c_1$ . The other compatibility constraint,  $c_2 = C(v_1, v_2, v_3)$ , is transformed into

$$\begin{aligned} c_{2R} &= (D_{1R} \times D_{2R} \times D_{3R}) - \overline{c_2} = (D_1 \times D_2 \times (D_3 \cup \{N\})) - ((D_1 \times D_2 \times D_3) - c_2) \\ &= c_2 \cup (D_1 \times D_2 \times \{N\}) = c_2 \cup \{(acN) (adN) (bcN) (bdN)\} \\ &= \{(bce) (bcf) (acf) (bde) (ade) (bdf) (adf)\} \end{aligned}$$

△

## Reformulation of Activity Constraints

Inclusion and exclusion activity constraints are reformulated as regular constraints defined on variables that include the target variable and the variables on which activation conditions are defined. The approach we take to express this reformulation is to determine what values of the target variable go with what tuples of the activation condition constraint.

Given an activity constraint  $a$  with activation condition  $a_{cond}$  and target variable  $v_t$ , reformulation  $a_R$  of  $a$  is defined on  $a_{cond}$ 's variables,  $var(a_{cond}) = V_{cond} = \{v_{c_1}, \dots, v_{c_k}\}$ , and target variable  $v_t$ . The construction of the reformulated constraint has two parts. They result from partitioning all possible value combinations of the condition variables into:

- $T_{cond}$ , the set of tuples that satisfy the condition constraint, that is,  $T_{cond} = a_{cond}$ , and
- $\overline{T_{cond}}$ , the set of tuples that invalidate the condition constraint, that is, the complement of  $T_{cond}$  with regard to the Cartesian product of the condition variable domains:

$$\overline{T_{cond}} = D_{v_{c_1}R} \times \dots \times D_{v_{c_k}R} - T_{cond}.$$

The allowed tuples of the reformulation  $a_R$  are, consequently, partitioned into tuples that extend  $T_{cond}$  and tuples that extend  $\overline{T_{cond}}$  with consistent values at  $v_t$  depending on the type of activation, of inclusion or exclusion, imposed on  $v_t$ . Next we present these two different reformulations separately.

**Reformulation of exclusion activity constraints.** If  $a$  is an exclusion activity constraint which is checked for some instantiation of its condition variables, then the target

variable is excluded if the condition constraint holds. In the reformulation  $a_R$ , we say that  $T_{cond}$  tuples are consistent with null values for  $v_t$ , and the set  $T_{cond} \times \{N\}$  can be added to the allowed tuples of  $a_R = A(V_{cond}, v_t)$ . If an instantiation violates  $T_{cond}$ , that instantiation represents a disallowed tuple that belongs to  $\overline{T_{cond}}$  and has no effect on the activity status of the target variable. We say that  $\overline{T_{cond}}$  tuples are consistent with any value of  $v_t$ . Note that  $v_t$  cannot be an initial variable, whose status is predefined and never affected by activity constraints. Therefore,  $D_{v_t}$  in the original CSP becomes  $D_{v_t R} = D_{v_t} \cup \{N\}$  in the reformulated problem (by algorithm 5.2). To express the consistency of  $\overline{T_{cond}}$  tuples with any value in  $D_{v_t R}$  we add  $\overline{T_{cond}} \times D_{v_t R}$  to  $a_R$ .

Combining the contribution of  $T_{cond}$  and  $\overline{T_{cond}}$  sets to the allowed tuples of  $a_R$  we obtain:

$$a_R = A(V_{cond}, v_t) = \{T_{cond} \times \{N\}\} \cup \{\overline{T_{cond}} \times D_{v_t R}\}$$

as shown in Algorithm 5.4 for reformulating exclusion activity constraints.

**Algorithm 5.4.** Adds to  $\mathcal{C}_R$  the reformulation of exclusion activity constraints in  $\mathcal{C}_A$ .

```

refExclusion( $\mathcal{C}_A, \mathcal{V}, \mathcal{D}_R, \mathcal{C}_R$ ) {
  for each ( $a \in \mathcal{C}_A$ ) {
    if ( $a$  is an exclusion activity constraint) {
      let  $V_{cond} = \{v_{c_1}, \dots, v_{c_k}\} \in \mathcal{V}$  be  $a$ 's condition variables
      let  $v_t \in \mathcal{V}$  be  $a$ 's target variable
      let  $T_{cond}$  be the allowed tuples of  $a$ 's activation condition
       $\overline{T_{cond}} \leftarrow D_{v_{c_1} R} \times \dots \times D_{v_{c_k} R} - T_{cond}$  // disallowed tuples of  $a$ 's activation condition
      let  $a_R$  be an empty constraint defined on  $V_{cond} \cup \{v_t\}$ 
       $a_R \leftarrow (T_{cond} \times \{N\}) \cup (\overline{T_{cond}} \times D_{v_t R})$ 
       $\mathcal{C}_R \leftarrow \mathcal{C}_R \cup \{a_R\}$ 
    } // end if
  } // end for
} // end refExclusion()

```

The same transformation can be obtained by stating the disallowed tuples in  $a_R$ , that is,  $a_R^{disallowed} = T_{cond} \times (D_{v_t R} - \{N\})$ . In other words, the only value assignments that  $a_R$  finds inconsistent are those which combine the allowed tuples of the activation condition with non-null values at  $v_t$ .

The transformation is illustrated on the following example, derived from Example 10.

**Example 12.** Since  $\mathcal{P}_1$  in Example 10 has only inclusion activity constraints, we construct  $\mathcal{P}_2$ , which modifies  $\mathcal{P}_1$  by adding an exclusion activity constraint  $a_2$  as shown in Figure 5-ii.

$$\begin{aligned}
\mathcal{P}_2 &= \langle \mathcal{V}_2, \mathcal{D}_2, \mathcal{V}_{I2}, \mathcal{C}_{C2}, \mathcal{C}_{A2} \rangle \\
\mathcal{V}_2 &= \mathcal{V}_1 = \{v_1, v_2, v_3\} \\
\mathcal{D}_2 &= \mathcal{D}_1 = \{D_1, D_2, D_3\} \\
\mathcal{V}_{I2} &= \mathcal{V}_{I1} = \{v_1, v_2\} \\
\mathcal{C}_{C2} &= \mathcal{C}_{C1} = \{c_1, c_2\} \\
\mathcal{C}_{A2} &= \mathcal{C}_{A1} \cup \{a_2\} = \{a_1, a_2\} \\
a_1 &= A(v_1, v_3) : v_1 = b \xrightarrow{\text{incl}} v_3 \\
a_2 &= A(v_2, v_3) : v_2 = c \xrightarrow{\text{excl}} v_3 \\
\text{sol}(\mathcal{P}_2) &= \{\{v_1 = a, v_2 = d\}\}
\end{aligned}$$

Figure 5-ii: Simple conditional CSP example  $\mathcal{P}_2$  with  $a_2$  exclusion activity constraint

The reformulation  $a_{2R}$  is defined on the condition variable  $v_2$  and target variable  $v_3$ . The activation condition constraint has the allowed value assignment  $v_2 = c$ . This condition variable instantiation is consistent with  $v_3 = N$  and stands for excluding  $v_3$  from partial solutions which satisfy  $v_2 = c$ . The other value assignment which accounts for the complement of the activation condition,  $v_2 = d$ , goes with all the domain values of variable  $v_3$ ,  $\{e, f, N\}$ . The condition variable instantiation  $v_2 = d$  stands for not restricting in any way  $v_3$ 's participation to solutions. Thus,

$$a_{2R} = A(v_2, v_3) = \{(c\ N)\ (d\ e)\ (d\ f)\ (d\ N)\}$$

△

**Reformulation of inclusion activity constraints.** We consider now the case of reformulating inclusion activity constraints.

Given an inclusion activity constraint  $a$ , we first construct  $a_R$ 's allowed tuples induced by  $T_{cond}$ . If all condition variables are active and the condition constraint holds, then the target variable is made active. In the reformulated constraint, the same behavior is obtained by making consistent the condition constraint allowed tuples,  $T_{cond}$ , with the non-null values of the target variable. Since  $v_t$  is not an initial variable, its domain in the original problem  $D_{v_t}$  becomes  $D_{v_tR} = D_{v_t} \cup \{N\}$  in the transformed null-based problem. Combining  $T_{cond}$  with  $v_t$ 's non-null values and adding the resulting tuples to  $a_R$ , we obtain  $T_{cond} \times (D_{v_tR} - \{N\}) \subset a_R$ . Note that another way to view this transformation is to disallow

the combination between the condition constraint tuples and the null value of the target variable, in which case we say  $T_{cond} \times \{N\} \not\subset a_R$ .

Second, we examine the relationship between  $\overline{T_{cond}}$  and  $v_t$ 's value domain  $D_{v_t R}$ . This relationship captures what happens with  $v_t$ 's activity status if  $T_{cond}$  does not hold, that is,  $\overline{T_{cond}}$  is true. There are two cases:

- *Single activations.*  $v_t$  is uniquely activated by  $a$ . In this case, if  $T_{cond}$  does not hold,  $v_t$  cannot be active. Its status remains undefined or excluded, and it cannot participate in solutions that do not satisfy  $T_{cond}$ .
- *Cluster activations.*  $v_t$  has other inclusion activations besides  $a$ . In this case, if none of the cluster activation conditions holds, then  $v_t$  is not active. In contrast to the single activation rule, cluster activation reformulation has to capture the interdependence of the cluster activation conditions. It is their interplay, rather than their independent contributions, that defines the reformulation. We will see later in the section that the cluster activation case is amended by an important assumption, that is, activations in a cluster do not form activity cycles.

In the rest of this subsection we present the single activation transformation: it directly reformulates an inclusion activity constraint that uniquely activates a target into an equivalent regular constraint. The algorithm *refSingleInclusion* implements this transformation. Its application to a sample problem is shown in Example 13. Cluster activations are considered in the next section.

Given an inclusion activity constraint,  $a$ , which solely activates the target  $v_t$  and has its condition constraint,  $T_{cond}$ , satisfied by some instantiation, then  $v_t$  is made active. In the reformulation  $a_R$ ,  $T_{cond}$  tuples are consistent with non-null values of  $v_t$ . Thus,  $T_{cond} \times (D_{v_t R} - \{N\})$  are the allowed tuples added to  $a_R$ . If  $T_{cond}$  does not hold, under the single activation assumption,  $v_t$  cannot be active and  $\overline{T_{cond}}$  tuples are consistent with null value at  $v_t$ . Thus,  $\{\overline{T_{cond}} \times \{N\}\}$  are the allowed tuples that complete the reformulation of  $a_R$ .

The algorithm *refSingleInclusion* (Algorithm 5.5) has a structure identical to *refExclusion*. It differs by the transformation rule that constructs  $a_R$ .

**Algorithm 5.5.** Adds to  $C_R$  the reformulation of single activations in  $C_A$ : inclusion activity constraints that solely activate target variables.

```

refSingleInclusion( $C_A, \mathcal{V}, \mathcal{D}_R, C_R$ ) {
  for each ( $a \in C_A$ ) {
    if ( $a$  is an inclusion activity constraint) {
      let  $V_{cond} = \{v_{c_1}, \dots, v_{c_k}\} \in \mathcal{V}$  be  $a$ 's condition variables
      let  $v_t \in \mathcal{V}$  be  $a$ 's target variable
      let  $T_{cond}$  be the allowed tuples of  $a$ 's activation condition
       $\overline{T_{cond}} \leftarrow D_{v_{c_1}R} \times \dots \times D_{v_{c_k}R} - T_{cond}$  // disallowed tuples of  $a$ 's activation condition
      let  $a_R$  be an empty constraint defined on  $V_{cond} \cup \{v_t\}$ 
       $a_R \leftarrow \{T_{cond} \times (D_{v_tR} - \{N\})\} \cup \{\overline{T_{cond}} \times \{N\}\}$ 
       $C_R \leftarrow C_R \cup \{a_R\}$ 
    } // end if
  } // end for
} // end refSingleInclusion()

```

**Example 13.** We generate the null-based reformulated CSP  $\mathcal{P}_{1R}$  from the original conditional CSP  $\mathcal{P}_1$  given in Example 10. We include the transformation of variable domains and compatibility constraints as shown in Example 11.

$$\begin{aligned}
\mathcal{P}_{R1} &= \langle \mathcal{V}_{R1}, \mathcal{D}_{R1}, C_{R1} \rangle \\
\mathcal{V}_{R1} &= \mathcal{V}_1 = \{v_1, v_2, v_3\} \\
\mathcal{D}_{R1} &= \{D_{1R}, D_{2R}, D_{3R}\} \\
&\quad D_{1R} = D_1 = \{a, b\}, D_{2R} = D_2 = \{c, d\}, D_{3R} = \{e, f, N\} \\
C_{R1} &= \{c_{1R}, c_{2R}, a_{1R}\} \\
&\quad c_{1R} = c_1 = C(v_1, v_2) = \{(a, d) (b, c)\} \\
&\quad c_{2R} = C(v_1, v_2, v_3) = \\
&\quad \quad \{(a, c, f) (b, c, e) (b, c, f) (a, d, e) (a, d, f) (b, d, e) (b, d, f) (a, c, N) (b, c, N) (a, d, N) (b, d, N)\} \\
&\quad a_{1R} = C(v_1, v_3) = \{(b, e) (b, f) (a, N)\} \\
sol(\mathcal{P}_{R1}) &= \{\{v_1 = a, v_2 = d, v_3 = N\}, \{v_1 = b, v_2 = c, v_3 = e\}, \{v_1 = b, v_2 = c, v_3 = f\}\}
\end{aligned}$$

Figure 5-iii: Reformulation  $\mathcal{P}_{R1}$  of the problem example  $\mathcal{P}_1$  in Figure 5-i shows the reformulation of single activation inclusion constraints.

The null-based reformulated CSP obtained from the conditional CSP in Example 10 is shown in Figure 5-iii.

The example has a single activity constraint  $a_1 = A(v_1, v_3) : v_1 = b \xrightarrow{\text{incl}} v_3$ . The activation condition  $v_1 = b$  represents the unary constraint  $T_{cond}(v_1) = \{(b)\}$ . The target variable is  $v_3$  with value domain  $D_{3R} = \{e, f, N\}$ . The reformulated constraint  $a_{1R}$  allows value combinations between the condition constraint tuples and non-null values  $e$  and  $f$  for  $v_3$ :  $\{(b, e) (b, f)\} \subset a_{1R}$ . Now we have to consider the allowed value pairs with which  $\overline{T_{cond}(v_1)} = \{(a)\}$  contributes to  $a_R$ . That is the pair  $(a, N)$ , which says that  $v_3$  cannot be

active if the condition  $v_1 = b$  fails. Thus,  $a_{1R} = \{(be) (bf) (aN)\}$ .

△

### 5.2.2 (Acyclic) Cluster Activations

Gelle observes that cluster activations can be reformulated by combining them first into one activity constraint, which is then transformed using the transformation of a single activation. She indicates that this approach has the disadvantage of not allowing an incremental introduction of inclusion activity constraints during problem reformulation. The drawback occurs when a cluster of activity constraints,  $A$ , that target the same variable, has been reformulated as  $a_R$ . If a new inclusion activity constraint,  $a$ , is added to the original problem, the reformulation procedure has to: discard  $a_R$ , rebuild the cluster  $A$  such that it contains  $a$ , and compute a new reformulation for  $A$ .

Although Gelle does not formally define how cluster activations are, in the general case, reformulated, she uses a simple example to illustrate how the single activation rule fails to correctly transform cluster activations. In the following, we use the same example to introduce the formalism for cluster activation reformulation and the algorithm *refClusterInclusion* that we developed to implement this formalism.

We discover that the clustering idea does not work if cluster activations have activity cycles. We recall from Chapter 2 the definition of an *activity cycle* in an inclusion activity graph. The inclusion activity graph of a conditional CSP has as directed edges inclusion activity constraints. An activity cycle is then a directed graph path of variables  $v_0, \dots, v_k$  such that  $v_0$  and  $v_k$  are the same variable and there are at least two directed edges (inclusion activity constraints) in the path. The assumption under which the cluster activation transformation works is that the activity graph has no cycles. We postpone the presentation of the cycle activity problem for the next section, in which we give some examples and a reformulation algorithm that handles cycles.

We conclude this section with an incremental version, the *refIncrementalInclusion* algorithm, that updates the reformulation locally without knowing in advance the com-

position of the acyclic activity constraint clusters. Note that all the other reformulation algorithms presented so far, *refDomains*, *refCompatibility*, and *refExclusion*, are incremental in nature and do not pose the problem of knowing beforehand the entire set of elements to be reformulated.

**Example 14.** Gelle's problem example  $\mathcal{P}_3$  modifies problem  $\mathcal{P}_1$  with the addition of the activity constraint  $a_2 = A(v_5, v_3) : v_5 = i \xrightarrow{\text{incl}} v_3$ , which activates the same variable as  $a_1$ .  $a_2$ 's condition variable,  $v_5$ , is a new initial active variable, whose values are  $D_5 = \{i, j\}$  (Figure 5-iv, left side).

$$\begin{array}{ll}
 \mathcal{P}_3 = \langle \mathcal{V}_3, \mathcal{D}_3, \mathcal{V}_{I3}, \mathcal{C}_{C3}, \mathcal{C}_{A3}, \rangle & \mathcal{P}_{R3} = \langle \mathcal{V}_{R3}, \mathcal{D}_{R3}, \mathcal{C}_{R3} \rangle \\
 \mathcal{V}_3 = \{v_1, v_2, v_3, v_5\} & \mathcal{V}_{R3} = \mathcal{V}_3 = \{v_1, v_2, v_3, v_5\} \\
 \mathcal{D}_3 = \{D_1, D_2, D_3, D_5\} & \mathcal{D}_{R3} = \{D_{1R}, D_{2R}, D_{3R}, D_{5R}\} \\
 D_1 = \{a, b\}, D_2 = \{c, d\}, & D_{1R} = D_1 = \{a, b\}, D_{2R} = D_2 = \{c, d\}, \\
 D_3 = \{e, f\}, D_5 = \{i, j\} & D_{3R} = \{e, f, N\}, D_{5R} = D_5 = \{i, j\} \\
 \mathcal{V}_{I3} = \{v_1, v_2, v_5\} & (bde)(bdf)(acN)(bcN)(adN) \\
 \mathcal{C}_{C3} = \{c_1, c_2\} & (bdN)\} \\
 c_1 = C(v_1, v_2) = \{(ad)(bc)\} & \mathcal{C}_{R3} = \{c_{1R}, c_{2R}, a_{12R}\} \\
 c_2 = C(v_1, v_2, v_3) = & c_{1R} = c_1 = C(v_1, v_2) = \{(ad)(bc)\} \\
 \{(acf)(bce)(bcf)(ade)(adf) & c_{2R} = C(v_1, v_2, v_3) = \\
 (bde)(bdf)\} & \{(acf)(bce)(bcf)(ade)(adf) \\
 \mathcal{C}_{A3} = \{a_1, a_2\} & (bde)(bdf)(acN)(bcN)(adN) \\
 a_1 = A(v_1, v_3) : v_1 = b \rightarrow v_3 & (bdN)\} \\
 a_2 = A(v_5, v_3) : v_5 = i \rightarrow v_3 & a_{12R} = C(v_1, v_5, v_3) = \\
 & \{(bie)(bif)(bje)(bjf)(aie)(aje) \\
 & (ajN)\} \\
 \\
 sol(\mathcal{P}_3) = & \{\{v_1 = a, v_2 = d\}, \{v_1 = b, v_2 = c, v_3 = e, v_5 = i\}, \\
 & \{v_1 = b, v_2 = c, v_3 = e, v_5 = j\}, \{v_1 = b, v_2 = c, v_3 = f\}\} \\
 sol(\mathcal{P}_{R3}) = & \{\{v_1 = a, v_2 = d, v_3 = N, v_5 = N\}, \{v_1 = b, v_2 = c, v_3 = e, v_5 = i\}, \\
 & \{v_1 = b, v_2 = c, v_3 = e, v_5 = j\}, \{v_1 = b, v_2 = c, v_3 = f, v_5 = N\}\}
 \end{array}$$

Figure 5-iv: (Left) Conditional CSP example  $\mathcal{P}_3$  with cluster activations. (Right) Reformulation  $\mathcal{P}_{R3}$  of  $\mathcal{P}_3$ . (Bottom) Problem solution set.

If we independently reformulate  $a_1$  and  $a_2$  using the single activation transformation for each of them, we obtain:

$$\begin{aligned}
 a_{1R} &= C(v_1, v_3) = \{(be)(bf)(aN)\} \\
 a_{2R} &= C(v_5, v_3) = \{(ie)(if)(jN)\}
 \end{aligned}$$

This transformation is incorrect, since it invalidates value assignments that account for satisfying either the  $v_1 = b$  or  $v_5 = i$  activation condition, but not both, and a non-null value for target variable  $v_3$ . Indeed, if, for example,  $v_1 = b$  and  $v_5 = j$ , the  $v_3$  activity status is set by  $a_1$  to active, and it is not influenced in any way by  $a_2$ 's failure to activate it. It means that  $\{v_5 = j, v_3 = N\} \in a_{2R}$  violates the instantiations  $\{v_1 = b, v_5 = j, v_3 = e\}$  and  $\{v_1 = b, v_5 = j, v_3 = f\}$ . This suggests that  $a_1$  and  $a_2$  are interdependent and can be combined in one constraint,  $a_{12}$ , as follows:

$$a_{12} = A(v_1, v_5, v_3) : (v_1 = b \vee v_5 = i) \xrightarrow{\text{incl}} v_3$$

The reformulation of  $a_{12R}$  is the result of the single activation transformation as implemented in the *refSingleInclusion* algorithm. The constraint  $a_{12R}$  allows that:

1. Either  $v_1 = b$  or  $v_5 = i$  be consistent with  $v_3$ 's non-null values,  $D_{v_3R} - \{N\} = \{e, f\}$ , regardless of the value assignment of the other condition variable:  $v_5$ , or  $v_1$ , respectively, and
2. Only  $(v_1 = a \wedge v_5 = j)$ , the complement of the  $a_{12}$  activation condition  $(v_1 = b \vee v_5 = i)$ , restricts the value assignment of  $v_3$  to  $N$ .

The computation of  $a_{12R}$  is:

$$\begin{aligned} a_{12R} &= C(v_1, v_5, v_3) \\ &= \{\{b\} \times D_{v_5R} \times (D_{v_3R} - \{N\})\} \cup \{D_{v_1R} \times \{i\} \times (D_{v_3R} - \{N\})\} \cup \{(a \ j \ N)\} \\ &= \{(b \ i \ e) \ (b \ i \ f) \ (b \ j \ e) \ (b \ j \ f) \ (a \ i \ e) \ (a \ i \ f) \ (a \ j \ N)\} \end{aligned}$$

△

In the general case where we do not restrict activation conditions to unary constraints, given two activity constraints of inclusion,  $a_i$  and  $a_j$ , of arity  $l$  and  $m$ , respectively, that activate the same variable  $v_t$

$$\begin{aligned} a_i &= A(v_{i_1}, \dots, v_{i_l}, v_t) : T_{cond_i} \xrightarrow{\text{incl}} v_t \\ a_j &= A(v_{j_1}, \dots, v_{j_m}, v_t) : T_{cond_j} \xrightarrow{\text{incl}} v_t \end{aligned}$$

with  $\text{var}(T_{\text{cond}_i}) = \{v_{i_1}, \dots, v_{i_l}\} = V_i$  and  $\text{var}(T_{\text{cond}_j}) = \{v_{j_1}, \dots, v_{j_m}\} = V_j$ , we construct an equivalent single activity constraint  $a_{ij}$ :

$$a_{ij} = A(v_{c_1}, \dots, v_{c_n}, v_t) : T_{\text{cond}} \xrightarrow{\text{incl}} v_t$$

where

$$\begin{aligned} T_{\text{cond}} &= T_{\text{cond}_i} \vee T_{\text{cond}_j} \\ \text{var}(T_{\text{cond}}) &= \{v_{i_1}, \dots, v_{i_l}\} \cup \{v_{j_1}, \dots, v_{j_m}\} = V_i \cup V_j = \{v_{c_1}, \dots, v_{c_n}\} = V_{\text{cond}} \\ |V_{\text{cond}}| &= n, 1 \leq n \leq l + m \\ a_{ij} \text{ arity} &\text{ is } n + 1 \end{aligned}$$

The reformulation  $a_{ijR}$  of  $a_{ij}$  is defined on the same set of variables as  $a_{ij}$

$$a_{ijR} = C(V_{\text{cond}} \cup \{v_t\})$$

and is made of two categories of allowed tuples.

In the first category we have tuples that are derived from  $T_{\text{cond}}$  and contribute to  $a_{ijR}$  by binding non-null values of  $v_t$  when either  $T_{\text{cond}_i}$  or  $T_{\text{cond}_j}$  or both constraints are true, regardless of the values of the other variables in  $V_{\text{cond}}$ . We denote these tuples of  $n + 1$  arity by  $E_{v_t}$ . They enforce a necessary condition for activating  $v_t$ .

$$E_{v_t} = T_{\text{cond}} \times (D_{v_tR} - \{N\})$$

$T_{\text{cond}}$  tuples are defined on  $V_{\text{cond}} = V_i \cup V_j$ . The set  $T_{\text{cond}}$  is computed from the union of  $n$ -ary tuples that extend  $T_{\text{cond}_i}$  and  $T_{\text{cond}_j}$  through the Cartesian product of the reformulated domains of variables in  $V_{\text{cond}}$  as follows: the extension of the  $T_{\text{cond}_i}$  tuples to  $n$ -ary tuples has all values in the reformulated domains of variables  $V_j$  that are not in  $V_i$ ; similarity, the extension of the  $T_{\text{cond}_j}$  tuples to  $n$ -ary tuples has all values in the reformulated domains of variables in  $V_j$  that are not in  $V_i$ . Thus:

$$T_{\text{cond}} = T_{\text{cond}_i} \times \prod_{v \in (V_j - V_i)} D_{vR} \cup T_{\text{cond}_j} \times \prod_{v \in (V_i - V_j)} D_{vR}.$$

The second category of tuples in the reformulation  $a_{ijR}$  represents tuples that extend  $\overline{T_{\text{cond}}}$  and contribute to  $a_{ijR}$  by making consistent the  $N$  value for  $v_t$ . We denote these

tuples of  $n + 1$  arity by  $P_{v_t}$ . They have the role of a sufficient condition that limits  $v_t$ 's activation to  $a_i$  and  $a_j$  exclusively:

$$P_{v_t} = \overline{T_{cond}} \times \{N\}$$

The computation of  $\overline{T_{cond}}$  from  $T_{cond}$  is immediate through a series of simple set operations that result in:

$$\overline{T_{cond}} = \overline{T_{cond_i}} \times \prod_{v \in (V_j - V_i)} D_{vR} \cap \overline{T_{cond_j}} \times \prod_{v \in (V_i - V_j)} D_{vR}$$

In order to generate correctly the tuples of the  $n$ -ary constraint  $T_{cond}$  and  $\overline{T_{cond}}$  defined on  $V_{cond}$ , we consider the set  $V_{cond}$  to be ordered. That is, there is an indexing function  $i_{V_{cond}} : V_{cond} \rightarrow N$ , which introduces a total order on  $V_{cond}$ . Based on this function, we define an *indexing product* operator,  $A \bigotimes_{V_{cond}} B$ ,  $A \cup B = V_{cond}$ , that transforms the set of tuples of the Cartesian product  $A \times B$  by reordering each tuple according to the indexing function on  $V_{cond}$ . The correct computation of  $T_{cond}$  and  $\overline{T_{cond}}$  is given by:

$$\begin{aligned} T_{cond} &= T_{cond_i} \bigotimes_{V_{cond}} \prod_{v \in (V_j - V_i)} D_{vR} \cup T_{cond_j} \bigotimes_{V_{cond}} \prod_{v \in (V_i - V_j)} D_{vR} \\ \overline{T_{cond}} &= \overline{T_{cond_i}} \bigotimes_{V_{cond}} \prod_{v \in (V_j - V_i)} D_{vR} \cap \overline{T_{cond_j}} \bigotimes_{V_{cond}} \prod_{v \in (V_i - V_j)} D_{vR} \end{aligned}$$

With these transformations completed,  $a_{ij}R$  is:

$$a_{ij}R = E_{v_t} \cup P_{v_t}$$

Note that for all variables  $v \in V_{cond}$ , the reformulation of their domains  $D_v$  is  $D_{vR} = D_v \cup \{N\}$  if  $v$  is not an initial variable.

This reformulation can be generalized for a cluster of arbitrary size  $m$  of multiple activity constraints of inclusion. The cluster constraints activate the target variable  $v_t$ .

**Given**

- A cluster  $A_{v_t} = \{a_{i_1}, \dots, a_{i_m}\}$ , with the activation conditions  $T_{cond_{i_1}}, \dots, T_{cond_{i_m}}$  defined on  $var(T_{cond_{i_1}}) = V_{i_1}, \dots, var(T_{cond_{i_m}}) = V_{i_m}$ ,

**We compute** reformulation

- $a_{mR} = C(V_{cond} \cup \{v_t\})$ , where  $V_{cond} = V_{i_1} \cup \dots \cup V_{i_m}$  and  $|V_{cond}| = n$

as follows:

$$a_{mR} = E_{mv_t} \cup P_{mv_t} = T_{cond} \times (D_{v_tR} - \{N\}) \cup \overline{T_{cond}} \times \{N\}$$

The representation of the  $n + 1$ -arity constraint  $a_{mR}$  includes  $E_{mv_t}$  constraints, which necessarily activate  $v_t$ , and  $P_{mv_t}$  constraints, which sufficiently activate  $v_t$  by limiting  $v_t$ 's activation to existing  $A_{v_t}$  constraints only. The computation of  $T_{cond}$  and  $\overline{T_{cond}}$  from  $m$  activation conditions  $T_{cond_i}$  is:

$$\begin{aligned} T_{cond} &= T_{cond_{i_1}} \bigotimes_{V_{cond}} \prod_{v \in (V_{cond} - V_{i_1})} D_{vR} \cup \dots \cup T_{cond_{i_m}} \bigotimes_{V_{cond}} \prod_{v \in (V_{cond} - V_{i_m})} D_{vR} \\ \overline{T_{cond}} &= \overline{T_{cond_{i_1}}} \bigotimes_{V_{cond}} \prod_{v \in (V_{cond} - V_{i_1})} D_{vR} \cap \dots \cap \overline{T_{cond_{i_m}}} \bigotimes_{V_{cond}} \prod_{v \in (V_{cond} - V_{i_m})} D_{vR} \end{aligned}$$

The algorithmic steps for reformulating cluster activations are described in the procedure *refClusterInclusion* (see Algorithm 5.6). The procedure builds clusters of activity constraints of inclusion,  $A_{v_t}$ , that target the same variable  $v_t$ .

The two sets  $T_{cond}$  and  $\overline{T_{cond}}$  are produced for each cluster  $A_{v_t}$  by calling the *ORActivity* procedure (Algorithm 5.7). These condition sets are then extended with non-null values and null values, respectively, for  $v_t$ , to form the  $E_{v_t}$  and  $P_{v_t}$  sets. The sets are then combined into the reformulation  $a_{mR}$  of the cluster  $A_{v_t}$ . This reformulation is added to the  $\mathcal{C}_R$  reformulation and the process continues until all clusters  $A_{v_t}$  are exhausted.

The *ORActivity* procedure computes the  $T_{cond}$  and  $\overline{T_{cond}}$  sets iteratively for each inclusion activity in the activation cluster  $A_{v_t}$ . In the loop, the procedure maintains two cumulative conditions:

- A cumulative condition that enforces  $v_t$ 's activation if any of the cluster conditions,  $T_i$ , holds. We call it  $cond_E$ . It incorporates the contribution of the  $T_i$  sets and it is initialized with the activation condition  $T_1$  of the first inclusion activity.
- A cumulative condition that prevents  $v_t$ 's activation if none of the cluster conditions holds. We call it  $cond_P$ . It incorporates the contribution of the  $\overline{T_i}$  sets and it is initialized with  $\overline{T_1}$ .

**Algorithm 5.6.** Adds to  $\mathcal{C}_R$  the reformulation of inclusion activity constraints. Cluster activations are handled by first (1) building clusters of activations,  $A_{v_t}$  corresponding to the same target variable  $v_t$ , and then by (2) translating each cluster into a single reformulated constraint. The latter step calls *ORActivity* algorithm (Algorithm 5.7).

```

refClusterInclusion( $\mathcal{C}_A, \mathcal{V}, \mathcal{D}_R, \mathcal{C}_R$ )
{
  //Build clusters of activations with the same target variable
   $\mathcal{A} \leftarrow \emptyset$  // the set of clusters
  //Initialize clusters
  for each ( $v_t \in \mathcal{V} - \mathcal{V}_T$ ) {
     $A_{v_t} \leftarrow \emptyset$  // cluster of inclusion constraints with target  $v_t$ 
     $V_{cond_{v_t}} \leftarrow \emptyset$  // set of all condition variables for all constraints in  $A_{v_t}$ 
     $\mathcal{A} \leftarrow \mathcal{A} \cup \{A_{v_t}\}$ 
  }
  //Populate clusters with their associated condition variable sets
  for each ( $a \in \mathcal{C}_A$ )
    if ( $a$  is an inclusion activity constraint) {
      let  $v_t$  be  $a$ 's target variable
       $A_{v_t} \leftarrow A_{v_t} \cup \{a\}$ 
       $V_{cond_{v_t}} \leftarrow V_{cond_{v_t}} \cup \{a\text{'s condition variables}\}$ 
    }
  //Build reformulated constraints from clusters
  for each  $A_{v_t} \in \mathcal{A}$ 
    ORActivity( $A_{v_t}, T_{cond}, \overline{T_{cond}}, V_{cond_{v_t}}$ )
    let  $a_{mR}$  be an empty constraint defined on  $V_{cond_{v_t}}$  and  $v_t$ 
    //Add contribution of allowed and disallowed tuples of activation condition
     $E_{v_t} \leftarrow T_{cond} \times (D_{v_t,R} - \{N\})$ 
     $P_{v_t} \leftarrow \overline{T_{cond}} \times \{N\}$ 
     $a_{mR} \leftarrow E_{v_t} \cup P_{v_t}$ 
     $\mathcal{C}_R \leftarrow \mathcal{C}_R \cup \{a_{mR}\}$ 
} // end refClusterInclusion()

```

## Incremental reformulation

The compositional structure of cluster activation transformation exhibits an incremental pattern that can be exploited to localize changes in the reformulated problem when the original cluster changes. We observe that with each additional activation that is conditioned by some  $T_{cond_i}$ , we add allowed tuples that make  $T_{cond_i}$  consistent with non-null values of  $v_t$  while with  $\overline{T_{cond_j}}$  we further restrict the set of tuples that are consistent with  $v_t$ 's null value. The obstacle to using this type of incrementality when computing the  $a_{mR}$  reformulation constraint of an  $m$ -size cluster is that the set of condition variables  $V_{cond}$  has to be known in advance. We propose to overcome this obstacle and construct reformulations incrementally

**Algorithm 5.7.** Computes  $T_{cond}$  and  $\overline{T_{cond}}$  for a cluster activation  $A_{v_t}$ .

```

ORActivity( $A_{v_t}; T_{cond}, \overline{T_{cond}}, V_{cond_{v_t}}$ )
{
  let  $a_1$  be the first activation of  $v_t$  in  $A_{v_t}$  such that
     $V_1$  are  $a_1$ 's condition variables
     $T_1$  are the condition's allowed tuples
     $\overline{T_1}$  are the condition's disallowed tuples
   $cond_V \leftarrow V_1$ 
   $cond_E \leftarrow T_1$ 
   $cond_P \leftarrow \overline{T_1}$ 
  for each (remaining  $a_i \in A_{v_t}$ ) {
     $cond_E \leftarrow cond_E \bigotimes_{cond_V \cup V_i} \prod_{v \in (V_i - cond_V)} D_{vR} \cup T_i \bigotimes_{cond_V \cup V_i} \prod_{v \in (cond_V - V_i)} D_{vR}$ 
     $cond_P \leftarrow cond_P \bigotimes_{cond_V \cup V_i} \prod_{v \in (V_i - cond_V)} D_{vR} \cap \overline{T_i} \bigotimes_{cond_V \cup V_i} \prod_{v \in (cond_V - V_i)} D_{vR}$ 
  } // end for
   $T_{cond} \leftarrow cond_E$ 
   $\overline{T_{cond}} \leftarrow cond_P$ 
   $V_{cond_{v_t}} \leftarrow cond_V$ 
} //end ORActivity()

```

with each additional activation that is added to the problem.

We know from reformulating single and cluster activations that the reformulation has tuples consistent with non-null values for the target variable, called the  $E_{v_t}$  constraint, and tuples consistent with null values at  $v_t$ , called the  $P_{v_t}$  constraint. The question is whether  $E_{v_t}$  and  $P_{v_t}$  can be constructed incrementally without knowing up front the set of condition variables from all activity constraints that activate  $v_t$ .

To illustrate the idea of incremental transformation, we start with a single activity constraint:

$$a_i = A(V_{cond_i}, v_t) : T_{cond_i} \xrightarrow{\text{incl}} v_t$$

and its reformulation:

$$a_{iR} = C(V_{cond_i}, v_t) = E_{v_t} \cup P_{v_t} = T_{cond_i} \times (D_{v_tR} - \{N\}) \cup \overline{T_{cond_i}} \times \{N\}$$

For the purpose of simplifying the notation of computational constructs that we will be using repeatedly in the rest of this section, we rewrite the reformulated constraint as follows:

$$C(V_{cond_i}, v_t) = C_{Xv_t} = E_{Xv_t} \cup P_{Xv_t} = X \overline{N} \cup \overline{X} N$$

where  $T_{cond_i} = X$ ,  $\overline{T_{cond_i}} = \overline{X}$ , and  $D_{v_t R} - \{N\} = \overline{N}$ .

Let us consider the addition of a new activity constraint:

$$a_j = A(V_{cond_j}, v_t) : T_{cond_j} \xrightarrow{\text{incl}} v_t$$

with  $V_{cond_i} \neq V_{cond_j}$ . We denote  $T_{cond_j}$  by  $Y$  and  $\overline{T_{cond_j}}$  by  $\overline{Y}$ . The reformulation of  $a_i$  and  $a_j$ , denoted by  $C(V_{cond_i} \cup V_{cond_j} \cup \{v_t\}) = C_{XYv_t}$ , combines the tuples in  $\overline{N}$  with either  $X$  or  $Y$  to compute the  $E_{XYv_t}$  constraint, and combines the value  $N$  with  $\overline{X}$  and  $\overline{Y}$  to compute the  $P_{XYv_t}$  constraint, as follows<sup>2</sup>:

$$\begin{aligned} C_{XYv_t} &= E_{XYv_t} \cup P_{XYv_t} = \{(XY \cup X\overline{Y} \cup \overline{X}Y) \times \overline{N}\} \cup \{\overline{X}\overline{Y}N\} \\ &= XY\overline{N} \cup X\overline{Y}\overline{N} \cup \overline{X}Y\overline{N} \cup \overline{X}\overline{Y}N \end{aligned}$$

$C_{XYv_t}$  can be computed from  $C_{Xv_t}$  in three steps:

1.  $E_{Xv_t}$ 's tuples in  $C_{Xv_t}$ ,  $X\overline{N}$ , are extended with  $Y$  and  $\overline{Y}$  to obtain  $XY\overline{N}$  and  $X\overline{Y}\overline{N}$
2. To complete the  $E_{XYv_t}$  constraint in  $C_{XYv_t}$ , a new set of tuples is computed from  $\overline{X}$  and  $Y$  to obtain  $\overline{X}Y\overline{N}$
3. The  $P_{Xv_t}$  constraint in  $C_{Xv_t}$ ,  $\overline{X}N$ , is extended with  $\overline{Y}$  to obtain  $P_{XYv_t}$  in  $C_{XYv_t}$ ,  $\overline{X}\overline{Y}N$ .

**Example 15.** We use problem example  $\mathcal{P}_3$  in Example 14 and assume that the multiple activations of variable  $v_3$ ,  $a_1$  and  $a_2$ , are reformulated incrementally. Thus,  $\mathcal{P}_{31}$  has only the  $a_1$  activity constraint in the original problem  $\mathcal{P}_3$ , and is reformulated into  $\mathcal{P}_{\mathcal{R}31}$ . We then add the  $a_2$  activity constraint to  $\mathcal{P}_{31}$  and obtain  $\mathcal{P}_{32} = \mathcal{P}_3$ , whose reformulation is  $\mathcal{P}_{\mathcal{R}32} = \mathcal{P}_{\mathcal{R}3}$ .

The first reformulation is shown in Figure 5-v (top), and has the set of constraints,  $\mathcal{C}_{\mathcal{C}31}$ , composed of  $\{c_{1R}, c_{2R}, a_{1R}\}$ .  $a_1$  is a single activity constraint that activates  $v_3$  when the activation condition  $X = \{v_1 = b\}$  holds. The reformulation  $a_{1R}$  combines non-null values

---

<sup>2</sup>We assume that the notation  $AB$ , where  $A$  and  $B$  are two constraints defined on  $V_A$  and  $V_B$  variables, uses the indexing product operator, i.e.,  $X \bigotimes_V Y$ , where  $V = V_A \cup V_B$ .

$$\begin{array}{ll}
\mathcal{P}_{31} = \langle \mathcal{V}_3, \mathcal{D}_3, \mathcal{V}_{I3}, \mathcal{C}_{C3}, \mathcal{C}_{A31}, \rangle & \mathcal{P}_{\mathcal{R}31} = \langle \mathcal{V}_{\mathcal{R}3}, \mathcal{D}_{\mathcal{R}3}, \mathcal{C}_{\mathcal{R}31} \rangle \\
\mathcal{C}_{A31} = \{a_1\} & \mathcal{C}_{\mathcal{R}31} = \{c_{1R}, c_{2R}, a_{1R}\} \\
a_1 = A(v_1, v_3) : v_1 = b \xrightarrow{\text{incl}} v_3 & a_{1R} = C(v_1, v_3) = X \overline{N} \cup \overline{X} \{N\} \\
X = \{v_1 = b\}, \overline{X} = \{v_1 = a\} & = \{b\} \times \{e, f\} \cup \{a\} \times \{N\} \\
\overline{N} = \{e, f\} & = \{(be) (bf) (aN)\}
\end{array}$$


---

$$\begin{array}{ll}
\mathcal{P}_{32} = \mathcal{P}_3 = \langle \mathcal{V}_3, \mathcal{D}_3, \mathcal{V}_{I3}, \mathcal{C}_{C3}, \mathcal{C}_{A3} \rangle & \mathcal{P}_{\mathcal{R}32} = \mathcal{P}_{\mathcal{R}3} = \langle \mathcal{V}_{\mathcal{R}3}, \mathcal{D}_{\mathcal{R}3}, \mathcal{C}_{\mathcal{R}3} \rangle \\
\mathcal{C}_{A3} = \mathcal{C}_{A31} \cup \{a_2\} & \mathcal{C}_{\mathcal{R}32} = \mathcal{C}_{\mathcal{R}3} = \{\mathcal{C}_{\mathcal{R}31} - \{a_{1R}\}\} \cup \{a_{12R}\} \\
a_2 = A(v_5, v_3) : v_5 = i \xrightarrow{\text{incl}} v_3 & = \{c_{1R}, c_{2R}, a_{12R}\} \\
Y = \{v_5 = i\}, \overline{Y} = \{v_5 = j\} & a_{12R} = C(v_1, v_5, v_3) \\
& = X(Y \cup \overline{Y}) \overline{N} \cup \overline{X} Y \overline{N} \cup \overline{X} \overline{Y} \{N\} \\
& = \{b\} \times \{i, j\} \times \{ef\} \cup \{a\} \times \{i\} \times \{ef\} \\
& \quad \cup \{a\} \times \{j\} \times \{N\} \\
& = (bie) (bif) (bje) (bjf) (aie) (aif) (ajN)
\end{array}$$

Figure 5-v: (Top) Reformulation of a single activity constraint of inclusion. (Bottom) Incremental reformulation of an additional cluster activation from a previous reformulation.

of the target variable  $v_3$ ,  $\{ef\}$ , with  $X$  and  $\{N\}$  with the complement of the activation condition,  $\overline{X}$ ,  $\{v_1 = a\}$ .

The second reformulation  $\mathcal{P}_{\mathcal{R}32}$  of  $\mathcal{P}_{32}$  (Figure 5-v (bottom)) is obtained from the first reformulation  $\mathcal{P}_{\mathcal{R}31}$  through a series of incremental changes that account for the incremental change of adding  $a_2$  to  $\mathcal{P}_{31}$ . Thus,  $a_{1R}$  is removed from the set  $\mathcal{C}_{\mathcal{R}31}$  of reformulated constraints of the first reformulation, and replaced with  $a_{12R}$ . The computation of  $a_{12R}$  takes into account  $a_1$ 's activation condition  $X = \{v_1 = b\}$  and its complement  $\overline{X} = \{v_1 = a\}$ , as well as the activation condition  $Y = \{v_5 = i\}$  and its complement  $\overline{Y} = \{v_5 = j\}$  of the additional activity constraint,  $a_2$ . The tuples consistent with  $\overline{N} = \{ef\}$  in  $a_{12R}$  extend  $X = \{v_1 = b\}$  with  $Y = \{v_5 = i\}$  and  $\overline{Y} = \{v_5 = j\}$ , and  $\overline{X} = \{v_1 = a\}$  with  $Y = \{v_5 = i\}$ . In this way, there is at least one activation condition that holds when the target variable takes a non-null value. The tuples consistent with  $\{N\}$  extend  $\overline{X} = \{v_1 = a\}$  with  $\overline{Y} = \{v_5 = j\}$ . Thus, neither  $v_1$  nor  $v_5$  are responsible for activating the target variable, as they are both instantiated with non-activation values.

In the general case,

**Given**

- a reformulated constraint  $C_{m_{v_t}}$  of  $m$  activity constraints,  $m \geq 1$ , that include  $v_t$ , and that is defined on  $\text{var}(C_{m_{v_t}}) = V_m \cup \{v_t\}$ , such that

$$C_{m_{v_t}} = T_m \overline{N} \cup \overline{T_m} \{N\}, \text{ where} \\ \overline{N} = D_{v_t, R} - \{N\}, \text{ and } V_m = \text{var}(T_{cond_m})$$

- an additional activity constraint of inclusion,  $a$ , defined as  $a = A(V_a \cup v_t) : Z \xrightarrow{\text{incl}} v_t$ , where  $V_a = \text{var}(Z)$

**We compute**

- a reformulated constraint  $C_{(m+1)_{v_t}}$ , from  $C_{m_{v_t}}$  and  $a$ , which is defined on  $V_{m+1} = V_m \cup V_a$  as follows:

$$\begin{aligned} C_{(m+1)_{v_t}} &= T_{m+1} \overline{N} \cup \overline{T_{m+1}} N \\ &= (T_m Z \cup T_m \overline{Z} \cup \overline{T_m} Z) \times \overline{N} \cup \overline{T_m} \overline{Z} N \\ &= T_m Z \overline{N} \cup T_m \overline{Z} \overline{N} \cup \overline{T_m} Z \overline{N} \cup \overline{T_m} \overline{Z} N \end{aligned}$$

Algorithm 5.8 shows the procedure *refIncrementalInclusion* that implements the incremental construction of reformulated constraints corresponding to cluster activations. For each non-initial variable,  $v_t$ , the algorithm maintains a reformulated constraint,  $C_{v_t}$ , that accounts for all inclusion activations of  $v_t$ . With the processing of each inclusion activity constraint  $a$  that targets  $v_t$ , the old reformulation  $C_{v_t}^{old}$  associated with  $v_t$ , if empty, is reused to produce a  $C_{v_t}^{new}$  reformulation. This new reformulation replaces the old one in  $\mathcal{C}_{\mathcal{R}}$ . If  $C_{v_t}^{old}$  is empty, the first reformulation of  $v_t$ 's cluster is computed from  $a$ 's elements.

### 5.2.3 Activity Cycles

The reformulation algorithms presented in the previous section work correctly for problems with acyclic activity graphs. In the following, we show on some examples why ac-

**Algorithm 5.8.** Adds to  $\mathcal{C}_{\mathcal{R}}$  the reformulation of inclusion activity constraints. Cluster activations are handled by incrementally generating reformulated constraints for each additional activation  $a$ .

```

refIncrementalInclusion( $\mathcal{C}_{\mathcal{A}}, \mathcal{V}, \mathcal{D}_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}$ ) {
  //Initialize with empty sets activation reformulations that might target non-null variables
  for each ( $v_i \in \mathcal{V} - \mathcal{V}_{\mathcal{I}}$ ) {
    let  $C_{v_i}$  be the reformulated constraint for all the activations of  $v_i$ 
     $C_{v_i} \leftarrow \emptyset$ 
  } //end for
  //Build or incrementally update  $C_{v_i}$  by processing
  //each activity constraint of inclusion,  $a$ , that targets  $v_i$ 
  for each ( $a \in \mathcal{C}_{\mathcal{A}}$ ) {
    let  $v_i$  be  $a$ 's target variable
    let  $\bar{N}$  be the non-null values of  $v_i$ 's domain
    let  $Z$  be the allowed tuples of  $a$ 's activation, and  $V_a$  be  $\text{var}(Z)$ 
    let  $\bar{Z}$  be the disallowed tuples of  $a$ 's activation condition
    //save the reformulation of previous activations of  $v_i$ 
     $C_{v_i}^{old} \leftarrow C_{v_i}$ 
    let  $V$  be the variables involved in  $C_{v_i}^{old}$ 
     $\mathcal{C}_{\mathcal{R}} \leftarrow \mathcal{C}_{\mathcal{R}} - C_{v_i}^{old}$ 
    let  $C_{v_i}^{new}$  be an empty constraint defined on  $V \cup V_a \cup \{v_i\}$  that
      reformulates  $a$  in the context of  $C_{v_i}^{old}$ 
    if ( $C_{v_i}^{old}$  is empty) {
       $C_{v_i}^{new} \leftarrow Z \bar{N} \cup \bar{Z} N$ 
    } // end if
    else {
      let  $T$  be  $C_{v_i}^{old}$ 's tuple set consistent with  $v_i$ 's non-null values
      let  $\bar{T}$  be  $C_{v_i}^{old}$ 's tuple set consistent with  $v_i$ 's  $N$  value
       $C_{v_i}^{new} \leftarrow T Z \bar{N} \cup T \bar{Z} \bar{N} \cup \bar{T} Z \bar{N} \cup \bar{T} \bar{Z} N$ 
    } //end else
     $\mathcal{C}_{\mathcal{R}} \leftarrow \mathcal{C}_{\mathcal{R}} \cup \{C_{v_i}^{new}\}$ 
  } // end for  $a$ 
} //end refIncrementalInclusion()

```

tivity cycles invalidate reformulation solutions produced with the *refClusterInclusion* algorithm. We then make changes that correct the reformulation rule of cluster activation. The result of this exercise is an idea worth exploring for a more general reformulation algorithm, unrestricted by activity cycles. We conclude the section with a new algorithm, *refGeneralClusterInclusion*, which produces a reformulation of inclusion activity constraints that might form activity cycles.

**Example 16.** Consider the simple conditional CSP problem  $\mathcal{P}_4$ , presented in Figure 5-vi (Left).

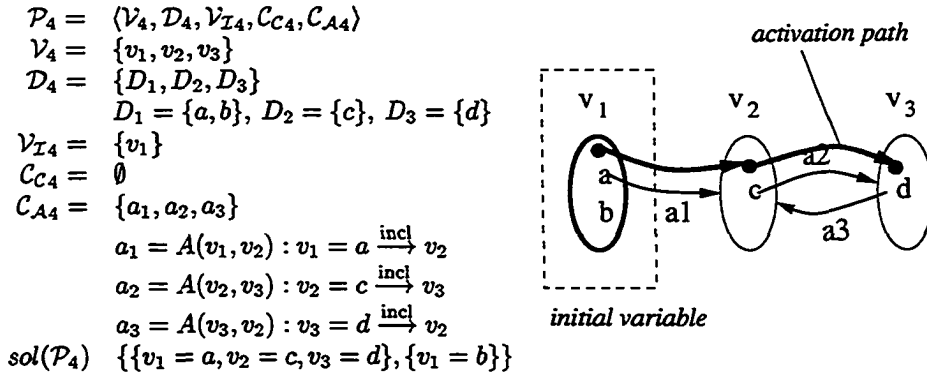


Figure 5-vi: Simple conditional CSP example,  $\mathcal{P}_4$ . (Left) Problem description. (Right)  $\mathcal{P}_4$ 's activity graph.

$\mathcal{P}_4$ 's inclusion activation graph has three variables and two inclusion activity constraints that form a cycle between variables  $v_2$  and  $v_3$ . If we reformulate the problem using the cluster activation algorithm, we obtain the problem  $\mathcal{P}_{41}$ , presented in Figure 5-vii (Top).

$ \begin{aligned} \mathcal{P}_{41} &= \langle \mathcal{V}_{41}, \mathcal{D}_{41}, \mathcal{C}_{41} \rangle \\ \mathcal{V}_{41} &= \{v_1, v_2, v_3\} \\ \mathcal{D}_{41} &= \{D_1, D_{2R}, D_{3R}\} \\ D_1 &= \{a, b\}, D_{2R} = \{c, N\}, D_{3R} = \{d, N\} \\ \mathcal{C}_{41} &= \{c_1, c_2\} \\ c_1 &= C(v_1, v_3, v_2) = \{(a d c) (a N c) (b d c) (b N N)\} \\ c_2 &= C(v_2, v_3) = \{(c d) (N N)\} \\ \text{sol}(\mathcal{P}_{41}) &= \{\{v_1 = a, v_2 = c, v_3 = d\}, \{v_1 = b, v_2 = c, v_3 = d\}, \{v_1 = b, v_2 = N, v_3 = N\}\} \end{aligned} $
$ \begin{aligned} \mathcal{P}_{42} \text{ reformulates } \mathcal{P}_4 \text{ by ignoring } a_3 \\ \mathcal{C}_{42} &= \{c'_1, c'_2\} \\ c'_1 &= C(v_1, v_2) = \{(a c) (b N)\} \\ c'_2 &= C(v_2, v_3) = \{(c d) (N N)\} \\ \text{sol}(\mathcal{P}_{42}) &= \{\{v_1 = a, v_2 = c, v_3 = d\}, \{v_1 = b, v_2 = N, v_3 = N\}\} \end{aligned} $

Figure 5-vii: (Top) Incorrect reformulation of  $\mathcal{P}_4$ . (Bottom) Correct reformulation of  $\mathcal{P}_4$ .

The second solution of the reformulated problem  $\mathcal{P}_{41}$ ,  $\{v_1 = b, v_2 = c, v_3 = d\}$ , is not a valid solution in the original problem  $\mathcal{P}_4$ . Indeed,  $v_1$  is a sole initial variable, which is the condition variable of only one activity constraint,  $a_1$ . If  $v_1 = b$ ,  $a_1$  does not activate  $v_2$ . The assignment  $v_1 = b$  cannot be extended to the solution  $\{v_1 = b, v_2 = c, v_3 = d\}$ , even if

$v_3$  activates  $v_2$  along another constraint,  $a_3$ .

The reformulated constraints,  $\mathcal{C}_{41}$ , do not capture the implicit requirement that: a variable, such as  $v_3$ , participate in the activation of another variable, such as  $v_2$ , if and only if  $v_3$  is already active.

The extraneous solution occurs because the non-initial variables  $v_2$  and  $v_3$  activate each other. To correct the problem, consider  $\mathcal{P}_4$ 's activity graph in Figure 5-vi (Right). It has only one activation path,  $\langle v_1, v_2, v_3 \rangle$ . Note that the activation path does not contain  $a_3$ . Therefore, the only variable that can activate  $v_3$  is  $v_2$ . It means that  $a_3$  is redundant. If the reformulation algorithm ignored  $a_3$  it would generate the correct solution set in Figure 5-vii (Bottom).

△

We observe that inclusion activity constraints that are not part of any activation path are redundant. Their removal eliminates activity cycles and, consequently, the reformulation produces correct solutions.

**Example 17.** The conditional CSP problem  $\mathcal{P}_5$  in Figure 5-viii (Left), has an activity graph with one cycle, formed by two activity constraints. However, both constraints participate in some activation path, Figure 5-viii (Right).

The activity graph has one cycle between  $v_3$  and  $v_4$ , which is given by  $a_3$ , on the first activation path, and  $a_4$ , on the second activation path. If we reformulate the problem using the cluster activation algorithm we obtain the problem  $\mathcal{P}_{51}$  in Figure 5-ix (Top).

The fourth solution of the reformulated problem  $\mathcal{P}_{51}$ ,  $\{v_1 = b, v_2 = d, v_3 = e, v_4 = f\}$ , is not a valid solution of the original problem  $\mathcal{P}_5$ . Similar to the incorrect solution in the previous example, even though the non-initial variables  $v_3$  and  $v_4$  activate each other, the initial variables  $v_1$  and  $v_2$  are not assigned values that activate  $v_3$  or  $v_4$ . The idea we used before to break the activity cycle does not apply anymore. Neither  $a_3$ , nor  $a_4$  is a redundant activity constraint, since they are part of the two activation paths (Figure 5-viii (Right)).

The example shows that cycles cannot be broken by simply discarding activity constraints. Again, the root of the problem is given by the observation that a variable must

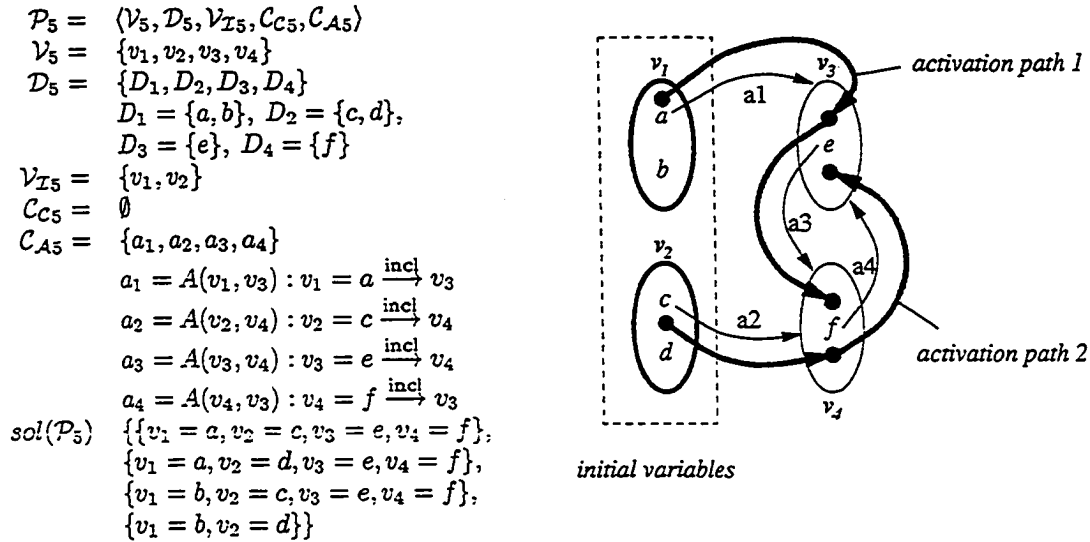


Figure 5-viii: Simple conditional CSP example,  $\mathcal{P}_5$ . (Left) Problem description. (Right)  $\mathcal{P}_5$ 's activity graph.

$$\begin{aligned} \mathcal{P}_{51} &= \langle \mathcal{V}_{51}, \mathcal{D}_{51}, \mathcal{C}_{C51} \rangle \\ \mathcal{V}_{51} &= \{v_1, v_2, v_3, v_4\} \\ \mathcal{D}_{51} &= \{D_1, D_2, D_{3R}, D_{4R}\} \\ D_1 &= \{a, b\}, D_2 = \{c, d\}, D_{3R} = \{e, N\}, D_{4R} = \{f, N\} \\ \mathcal{C}_{C51} &= \{c_1, c_2\} \\ c_1 &= C(v_1, v_4, v_3) = \{(a f e) (a N e) (b f c) (b N N)\} \\ c_2 &= C(v_2, v_3, v_4) = \{(c e f) (c N f) (d e f) (d N N)\} \\ \text{sol}(\mathcal{P}_{51}) &= \{ \{v_1 = a, v_2 = c, v_3 = e, v_4 = f\}, \{v_1 = a, v_2 = d, v_3 = e, v_4 = f\}, \\ &\quad \{v_1 = b, v_2 = c, v_3 = e, v_4 = f\}, \{v_1 = b, v_2 = d, v_3 = e, v_4 = f\}, \\ &\quad \{v_1 = b, v_2 = d, v_3 = N, v_4 = N\} \} \end{aligned}$$

Figure 5-ix: Incorrect reformulation of  $\mathcal{P}_5$

be active in order to trigger the activation of another variable. It is important, then, to determine what leads to the activation of each problem variable.

In  $\mathcal{P}_5$   $v_1$  and  $v_2$  are part of the initial variable set, and therefore, always active. The activity of  $v_3$  is controlled:

- Directly by the value of the initial (active) variable  $v_1$ , via  $a_1 : v_1 = a \xrightarrow{\text{incl}} v_3$ , or,
- Indirectly by the value of the initial (active) variable  $v_2$ , via  $a_2 : v_2 = c \xrightarrow{\text{incl}} v_4$ , as well as by the value of  $v_4$  via  $a_4 : v_4 = f \xrightarrow{\text{incl}} v_3$ . This is because  $v_4$  must be active in order to trigger  $a_4$ .

To summarize, the activity status of  $v_3$  is determined either by  $a_1$  or by  $a_2$  and  $a_4$ . Similarly, the activity status of  $v_4$  is determined either by  $a_2$  or by  $a_1$  and  $a_3$ .

Let us first reformulate the activation of  $v_3$ . We observe that the interplay among the activity constraints that control  $v_3$ , that is,  $a_1$ ,  $a_2$ , and  $a_4$ , is not a cluster activation of disjunctive single activations. Instead, it is a *disjunction* between  $a_1$  and the *conjunction* of  $a_2$  and  $a_4$ .

Calculating a reformulated constraint that corresponds to a conjunction of two activations,  $T_{cond_i} \wedge T_{cond_j}$ , is done similarly to the way we use to calculate the disjunction,  $T_{cond_i} \vee T_{cond_j}$ , of two cluster activations (Section 5.2.2). The reformulation of a conjunction of two activity constraints has

- $T_{cond}$  tuples that are consistent with non-null values of the target variable when  $T_{cond_i}$  and  $T_{cond_j}$  are true, regardless of the values in the other variables in  $V_{cond}$ , and
- $\overline{T_{cond}}$  tuples that are consistent with null values of the target variable.

These tuple sets have values from  $V_{cond} = V_i \cup V_j$  and are computed as follows:

$$\begin{aligned} V_{cond} &= V_i \cup V_j \\ T_{cond} &= T_{cond_i} \bigotimes_{V_{cond}} \prod_{v \in (V_j - V_i)} D_{vR} \cap T_{cond_j} \bigotimes_{V_{cond}} \prod_{v \in (V_i - V_j)} D_{vR} \\ \overline{T_{cond}} &= \overline{T_{cond_i}} \bigotimes_{V_{cond}} \prod_{v \in (V_j - V_i)} D_{vR} \cup \overline{T_{cond_j}} \bigotimes_{V_{cond}} \prod_{v \in (V_i - V_j)} D_{vR} \end{aligned}$$

To facilitate the representation of the reformulation of  $v_3$ 's activation, we find it useful to use a triplet notation that identifies three components for each activity constraint  $a_i$  that participates in a variable activation. These components are: the activation condition,  $T_{cond_i}$ , its complement,  $\overline{T_{cond_i}}$ , and the set of condition variables  $V_{cond_i}$  on which  $T_{cond_i}$  and  $\overline{T_{cond_i}}$  are defined. Thus,  $a_1$ ,  $a_2$ , and  $a_4$  inclusion activity constraints of  $v_3$  are represented as activations  $T_1$ ,  $T_2$ , and  $T_4$  as follows:

$$\begin{aligned} T_1 &: \langle T_{cond_1} = \{a\}, \overline{T_{cond_1}} = \{b\}, V_{cond_1} = \{v_1\} \rangle \\ T_2 &: \langle T_{cond_2} = \{c\}, \overline{T_{cond_2}} = \{d\}, V_{cond_2} = \{v_2\} \rangle \\ T_4 &: \langle T_{cond_4} = \{f\}, \overline{T_{cond_4}} = \{N\}, V_{cond_4} = \{v_4\} \rangle \end{aligned}$$

Before we derive the reformulation of  $v_3$ 's activation, we give the implementation of the *ANDActivity* procedure (Algorithm 5.9). It computes the reformulation  $\langle T_{cond}, \overline{T_{cond}}, V_{cond} \rangle$  of a conjunction over a set  $S$  of activations  $s_i : \langle T_i, \overline{T_i}, V_i \rangle$ . The algorithm design is similar to the one used in the implementation of the *ORActivity* procedure (Algorithm 5.7). The

**Algorithm 5.9.** *Computes  $\langle T_{cond}, \overline{T_{cond}}, V_{cond} \rangle$  from the conjunction of a set  $S$  of activations  $\langle T_i, \overline{T_i}, V_i \rangle$ .*

```

ANDActivity( $S; T_{cond}, \overline{T_{cond}}, V_{cond}$ )
{
  let  $s_1$  be the first activation in  $S$  such that
     $V_1$  are  $s_1$ 's condition variables
     $T_1$  are the condition's allowed tuples
     $\overline{T_1}$  are the condition's disallowed tuples
   $cond_V \leftarrow V_1$ 
   $cond_E \leftarrow T_1$ 
   $cond_P \leftarrow \overline{T_1}$ 
  for each (remaining  $s_i \in S$ ) {
     $cond_E \leftarrow cond_E \bigotimes_{cond_V \cup V_i} \prod_{v \in (V_i - cond_V)} D_{vR} \cap T_i \bigotimes_{cond_V \cup V_i} \prod_{v \in (cond_V - V_i)} D_{vR}$ 
     $cond_P \leftarrow cond_P \bigotimes_{cond_V \cup V_i} \prod_{v \in (V_i - cond_V)} D_{vR} \cup \overline{T_i} \bigotimes_{cond_V \cup V_i} \prod_{v \in (cond_V - V_i)} D_{vR}$ 
  } // end for
   $\overline{T_{cond}} \leftarrow cond_E$ 
   $T_{cond} \leftarrow cond_P$ 
   $V_{cond_{v_i}} \leftarrow cond_V$ 
} //end ANDActivity()

```

expression of the two conjunctive activations  $a_2$  and  $a_4$  that contribute to the activation of  $v_3$  is given by  $T_{cond_{24}} = T_{cond_2} \wedge T_{cond_4}$ .  $T_{24} : \langle T_{cond_{24}}, \overline{T_{cond_{24}}}, V_{cond_{24}} \rangle$  is then computed as follows:

$$\begin{aligned}
V_{cond_{24}} &= V_{cond_2} \cup V_{cond_4} = \{v_2, v_4\} \\
T_{cond_{24}} &= \{c\} \bigotimes_{\{v_2, v_4\} \ v \in \{v_4\}} \left( \prod_{v \in \{v_4\}} D_{vR} \right) \cap \{f\} \bigotimes_{\{v_2, v_4\} \ v \in \{v_2\}} \left( \prod_{v \in \{v_2\}} D_{vR} \right) \\
&= \{c\} \bigotimes_{\{v_2, v_4\}} \{f, N\} \cap \{f\} \bigotimes_{\{v_2, v_4\}} \{c, d\} \\
&= \{(c, f), (c, N)\} \cap \{(c, f), (d, f)\} \\
&= \{(c, f)\} \\
\overline{T_{cond_{24}}} &= \{d\} \bigotimes_{\{v_2, v_4\} \ v \in \{v_4\}} \left( \prod_{v \in \{v_4\}} D_{vR} \right) \cup \{N\} \bigotimes_{\{v_2, v_4\} \ v \in \{v_2\}} \left( \prod_{v \in \{v_2\}} D_{vR} \right) \\
&= \{d\} \bigotimes_{\{v_2, v_4\}} \{f, N\} \cup \{N\} \bigotimes_{\{v_2, v_4\}} \{c, d\} \\
&= \{(d, f), (d, N)\} \cup \{(c, N), (d, N)\} \\
&= \{(c, N), (d, f), (d, N)\}
\end{aligned}$$

Next, we calculate  $T_{cond_{124}} = T_{cond_1} \vee T_{cond_{24}}$ , the activity condition which governs the activation of variable  $v_3$ . Using the definition for calculating disjunction of two activity constraints, we obtain  $T_{124} : \langle T_{cond_{124}}, \overline{T_{cond_{124}}}, V_{cond_{124}} \rangle$ :

$$\begin{aligned}
V_{cond_{124}} &= V_{cond_1} \cup V_{cond_{24}} = \{v_1\} \cup \{v_2, v_4\} = \{v_1, v_2, v_4\} \\
T_{cond_{124}} &= \{a\} \bigotimes_{\{v_1, v_2, v_4\} \ v \in \{v_2, v_4\}} \left( \prod_{v \in \{v_2, v_4\}} D_{vR} \right) \cup \{(c, f)\} \bigotimes_{\{v_1, v_2, v_4\} \ v \in \{v_1\}} \left( \prod_{v \in \{v_1\}} D_{vR} \right) \\
&= \{a\} \bigotimes_{\{v_1, v_2, v_4\}} \{(c, f), (c, N), (d, f), (d, N)\} \cup \\
&\quad \{(c, f)\} \bigotimes_{\{v_1, v_2, v_4\}} \{a, b\} \\
&= \{(a, c, f), (a, c, N), (a, d, f), (a, d, N)\} \cup \\
&\quad \{(a, c, f), (b, c, f)\} \\
&= \{(a, c, f), (a, c, N), (a, d, f), (a, d, N), (b, c, f)\} \\
\overline{T_{cond_{124}}} &= \{b\} \bigotimes_{\{v_1, v_2, v_4\} \ v \in \{v_2, v_4\}} \left( \prod_{v \in \{v_2, v_4\}} D_{vR} \right) \cap \{(c, N), (d, f), (d, N)\} \bigotimes_{\{v_1, v_2, v_4\} \ v \in \{v_1\}} \left( \prod_{v \in \{v_1\}} D_{vR} \right) \\
&= \{b\} \bigotimes_{\{v_1, v_2, v_4\}} \{(c, f), (c, N), (d, f), (d, N)\} \cap \\
&\quad \{(c, N), (d, f), (d, N)\} \bigotimes_{\{v_1, v_2, v_4\}} \{a, b\} \\
&= \{(b, c, f), (b, c, N), (b, d, f), (b, d, N)\} \cap \\
&\quad \{(a, c, N), (a, d, f), (a, d, N), (b, c, N), (b, d, f), (b, d, N)\} \\
&= \{(b, c, N), (b, d, f), (b, d, N)\}
\end{aligned}$$

Similarly, we calculate  $T_{cond_{213}} = T_{cond_2} \vee T_{cond_{13}}$ , the activity condition which governs the activation of variable  $v_4$ , and we obtain  $T_{213} : \langle T_{cond_{213}}, \overline{T_{cond_{213}}}, V_{cond_{213}} \rangle$ :

$$\begin{aligned}
\mathcal{P}_{44} &= \langle \mathcal{V}_{44}, \mathcal{D}_{44}, \mathcal{C}_{44} \rangle \\
\mathcal{V}_{44} &= \{v_1, v_2, v_3, v_4\} \\
\mathcal{D}_{44} &= \{D_1, D_2, D_{3R}, D_{4R}\} \\
&\quad D_1 = \{a, b\}, D_2 = \{c, d\}, D_{3R} = \{e, N\}, D_{4R} = \{f, N\} \\
\mathcal{C}_{44} &= \{c_1, c_2\} \\
c_1 &= C(v_1, v_2, v_4, v_3) \\
&= \{(a, c, f, e), (a, c, N, e), (a, d, f, e), (a, d, N, e), (b, c, f, e), \\
&\quad (b, c, N, N), (b, d, f, N), (b, d, N, N)\} \\
c_2 &= C(v_2, v_1, v_3, v_4) \\
&= \{(c, a, e, f), (c, a, N, f), (c, b, e, f), (c, b, N, f), (d, a, e, f), \\
&\quad (d, a, N, N), (d, b, e, N), (d, b, N, N)\} \\
sol(\mathcal{P}_{44}) &= \{\{v_1 = a, v_2 = c, v_3 = e, v_4 = f\}, \{v_1 = b, v_2 = c, v_3 = e, v_4 = f\}, \\
&\quad \{v_1 = a, v_2 = d, v_3 = e, v_4 = f\}, \{v_1 = b, v_2 = d, v_3 = N, v_4 = N\}\}
\end{aligned}$$

Figure 5-x: Correct reformulation for  $\mathcal{P}_4$ 

$$\begin{aligned}
V_{cond_{213}} &= \{v_2, v_1, v_3\} \\
T_{cond_{213}} &= \{(c, a, e), (c, a, N), (c, b, e), (c, b, N), (d, a, e)\} \\
\overline{T}_{cond_{213}} &= \{(d, a, N), (d, b, e), (d, b, N)\}
\end{aligned}$$

The **correct** reformulation of  $\mathcal{P}_4$  is shown in Figure 5-x.

△

To provide the implementation of a general reformulation algorithm that handles activity cycles, we need a labeling function to “order” problem variables according to their level of activation.

The function sets the activation level of the initial variables to 0. Variables that cannot be reached by an activation path are labeled with -1. All the other variables are target variables,  $v$ , which participate in some inclusion activity constraints,  $a \in \mathcal{C}_A$ , and whose activation levels are computed from the activation levels of their condition variables,  $c = cond(a)$ , which have been already labeled, as follows:

$$\lambda(v) = \left( \min_{a \in \mathcal{C}_A | target(a)=v} \left( \max_{t \in cond(a)} \lambda(t) \right) \right) + 1$$

**Definition 18 (labeling function).** *Given an activity graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ , we define a labeling function  $\lambda : \mathcal{V} \rightarrow \mathbb{N}$ , such that:*

- $\lambda(v)$  is 0, if  $v$  is an initial variable, or

- $\lambda(v)$  is -1, if  $v$  is a non-initial variable and there is no inclusion activation path ending in  $v$ , or
- $\lambda(v) = ( \min_{a \in \mathcal{C}_A | \text{target}(a)=v} ( \max_{t \in \text{cond}(a)} \lambda(t) ) ) + 1$  if  $v$  is a non-initial variable and there is an inclusion activation path ending in  $v$ .

Using this definition, the *LabelActivityGraph* procedure computes the labels of the activation levels of all problem variables. Algorithm 5.10 shows the procedure *LabelActivityGraph*

**Algorithm 5.10.** We denote by  $\lambda(v)$  the label associated with variable  $v$ . The algorithm performs a breadth-first traversal of the inclusion activation graph and sequentially labels variables visited in each step.

```

LabelActivityGraph( $\mathcal{V}, \mathcal{V}_I, \mathcal{C}_A, \lambda$ )
{
  for each ( $v \in \mathcal{V}_I$ )  $\lambda(v) \leftarrow 0$ 
  for each ( $v \in \mathcal{V} - \mathcal{V}_I$ )  $\lambda(v) \leftarrow -1$ 
  let agenda be  $\mathcal{C}_A$ 
  let currLevel be an activation level set to 1
  while (agenda  $\neq \emptyset$ ) {
    let  $\mathcal{A}$  be agenda's activity constraints  $a$  s.t.  $\forall v \in \text{cond}(a), \lambda(v) \geq 0$ 
    if ( $\mathcal{A} = \emptyset$ ) break
    else {
      for each ( $a \in \mathcal{A}$ )
        if ( $\lambda(\text{target}(a)) = -1$ )
           $\lambda(\text{target}(a)) \leftarrow \text{currLevel}$ 
        agenda  $\leftarrow$  agenda  $- \mathcal{A}$ 
        currLevel  $\leftarrow$  currLevel + 1
    }
  } //end while
  if (agenda  $\neq \emptyset$ ) {
    // activity constraints left in the agenda will never trigger
     $\mathcal{C}_A \leftarrow \mathcal{C}_A - \text{agenda}$  //remove them from the problem
    // all variables with label -1 are unreachable
    for each ( $v \in \mathcal{V} - \mathcal{V}_I$ )
      if ( $\lambda[v] = -1$ ) //remove them from the problem
         $\mathcal{V} \leftarrow \mathcal{V} - \{v\}$ 
  }
}

```

which, given a conditional constraint satisfaction problem  $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ , assigns each variable  $v \in \mathcal{V}$  of the activation graph a label equal to  $\lambda(v)$ . In addition, inclusion activity constraints whose condition variables are never made active are removed from  $\mathcal{P}$ .

Also, variables unreachable via activation paths are removed from the problem too. The *LabelActivityGraph* algorithm is run prior to reformulating the problem components.

The reformulation algorithm that handles cycles in the inclusion activity constraints is *refGeneralClusterInclusion* (Algorithm 5.11). It is identical to the previous reformulation, *refClusterInclusion* (Algorithm 5.6), except for the call to *refOneCluster* (Algorithm 5.12), which replaces the *ORActivity* call and implements a more general cluster activation transformation.

**Algorithm 5.11.** *Generalizes refClusterInclusion (Algorithm 5.6), from which it differs by the boxed call. The transformation of each cluster into a single reformulated constraint is done by calling refOneCluster algorithm (Algorithm 5.12), which handles cycles in the inclusion activity constraints.*

```

refGeneralClusterInclusion( $\mathcal{C}_A, \mathcal{V}, \mathcal{D}_R, \mathcal{C}_R$ )
{
  //Build clusters of activations with the same target variable
   $\mathcal{A} \leftarrow \emptyset$  // the set of clusters
  //Initialize clusters
  for each ( $v_t \in \mathcal{V} - \mathcal{V}_I$ ) {
     $A_{v_t} \leftarrow \emptyset$  // cluster of inclusion constraints with target  $v_t$ 
     $V_{cond_{v_t}} \leftarrow \emptyset$  // set of all condition variables for all constraints in  $A_{v_t}$ 
     $\mathcal{A} \leftarrow \mathcal{A} \cup \{A_{v_t}\}$ 
  }
  //Populate clusters with their associated condition variable sets
  for each ( $a \in \mathcal{C}_A$ )
    if ( $a$  is an inclusion activity constraint) {
      let  $v_t$  be  $a$ 's target variable
       $A_{v_t} \leftarrow A_{v_t} \cup \{a\}$ 
       $V_{cond_{v_t}} \leftarrow V_{cond_{v_t}} \cup \{a\}$ 's condition variables
    }
  //Build reformulated constraints from clusters
  for each  $A_{v_t} \in \mathcal{A}$ 
    refOneCluster( $A_{v_t}, T_{cond}, \overline{T_{cond}}, V_{cond_{v_t}}$ )
    let  $a_{mR}$  be an empty constraint defined on  $V_{cond_{v_t}}$  and  $v_t$ 
    //Add contribution of allowed and disallowed tuples of activation condition
     $E_{v_t} \leftarrow T_{cond} \times (D_{v_t R} - \{N\})$ 
     $P_{v_t} \leftarrow \overline{T_{cond}} \times \{N\}$ 
     $a_{mR} \leftarrow E_{v_t} \cup P_{v_t}$ 
     $\mathcal{C}_R \leftarrow \mathcal{C}_R \cup \{a_{mR}\}$ 
  } // end refGeneralClusterInclusion()

```

The reformulation algorithm for an activation cluster that might have cycles, *refOneCluster*,

uses three procedures: *ORActivity* (Algorithm 5.7), *ANDActivity* (Algorithm 5.9), and *AlternateActivation* (Algorithm 5.13).

**Algorithm 5.12.** Computes the reformulation  $\langle T_{cond}, \overline{T_{cond}}, V_{cond} \rangle$  from a set of set activity constraints of inclusion,  $A_{v_t}$ , that target the same variable  $v_t$ .

```

refOneCluster( $A_{v_t}; T_{cond}, \overline{T_{cond}}, V_{cond}$ ) {
  let vtCluster be an initially empty set of activations  $c : \langle cT, \overline{cT}, cV \rangle$ 
  for each  $a \in A_{v_t}$  {
    let aSources be an initially empty set of activations  $s : \langle sT, \overline{sT}, sV \rangle$ 
    for each  $(v \in \mathcal{V} - \mathcal{V}_T)$  visited( $v$ )  $\leftarrow$  false
    visited( $v_t$ )  $\leftarrow$  true
    targetLevel  $\leftarrow$   $\lambda(v_t)$ 
    aIsDiscarded  $\leftarrow$  false
    for each (a's condition variable  $u$  such that  $\lambda(u) \geq \text{targetLevel}$ ) {
      if (AlternateActivation( $u, \lambda(v_t), sT, \overline{sT}, sV$ ) is false) {
        aIsDiscarded  $\leftarrow$  true; break
      }
      else
        aSources  $\leftarrow$  aSources  $\cup \langle sT, \overline{sT}, sV \rangle$ 
    } //end for  $s$ 
    if (aIsDiscarded is false)
      if (aSources is empty)
        vtCluster  $\leftarrow$  vtCluster  $\cup \langle T_{cond_a}, \overline{T_{cond_a}}, V_{cond_a} \rangle$ 
      else {
        aSources  $\leftarrow$  aSources  $\cup \langle T_{cond_a}, \overline{T_{cond_a}}, V_{cond_a} \rangle$ 
        ANDActivity(aSources,  $cT, \overline{cT}, cV$ )
        vtCluster  $\leftarrow$  vtCluster  $\cup \langle cT, \overline{cT}, cV \rangle$ 
      } //end else
    } //end for  $a$ 
    ORActivity(vtCluster,  $T_{cond}, \overline{T_{cond}}, V_{cond_{v_t}}$ )
  } //end refOneCluster

```

*ORActivity* is called to reformulate an activation cluster, let us call it *vtCluster*, and returns the final result of the *refOneCluster*,  $\langle T_{cond}, \overline{T_{cond}}, V_{cond} \rangle$ .

The cluster activation *vtCluster* passed to *ORActivity* is obtained from the set of inclusion activity constraints that target the same variable  $v_t$ , called  $A_{v_t}$ . To compute *vtCluster*, each inclusion activity constraint  $a \in A_{v_t}$  has its condition variables checked. If for some  $a$  there is at least one condition variable whose activation depends solely on the activation of  $a$ 's target variable, then  $a$  is part of an activity cycle and is discarded. Otherwise,  $a$ 's participation in the *vtCluster* is determined based on the activation level of

its condition variables. This leads to two cases.

In the first case, if all  $a$ 's condition variables have activation levels, as determined by the labeling function  $\lambda$ , lower than  $v_t$ 's activation level, then  $a$  participates in the construction of  $vtCluster$  directly, through its condition's allowed tuples,  $T_{cond_a}$ , and disallowed tuples,  $\overline{T_{cond_a}}$ :

$$vtCluster \leftarrow vtCluster \cup \langle T_{cond_a}, \overline{T_{cond_a}}, V_{cond_a} \rangle$$

In the second case, some of the  $a$ 's condition variables, let us call them  $u$ , have activation levels higher than  $\lambda(v_t)$ . Therefore, the *AlternateActivation* procedure is called on each  $u$  to find an alternate activation,  $\langle sT, \overline{sT}, sV \rangle$ , that does not pass through  $v_t$ . These alternate activations or reformulations are stored in the set  $aSources$  together with  $a$ 's activation condition,  $\langle T_{cond_a}, \overline{T_{cond_a}}, V_{cond_a} \rangle$ . The conjunction of the activations in  $aSources$  ensures that all condition variables are made active independently of and prior to  $v_t$ 's activation. Thus  $aSources$  is reformulated by calling *ANDActivity* procedure, which produces  $\langle cT, \overline{cT}, cV \rangle$  reformulation. This result is then added to  $vtCluster$ :

$$ANDActivity(aSources, cT, \overline{cT}, cV)$$

$$vtCluster \leftarrow vtCluster \cup \langle cT, \overline{cT}, cV \rangle$$

*AlternateActivation* (Algorithm 5.13) operates on the condition variables of some inclusion activity constraint  $a$ . The procedure finds reformulations of activations of condition variables such that the activations do come through  $a$ 's target variable. If such alternate activations are not found, *AlternateActivation* returns false, which means there is an activity cycle between the condition variable and its target variable. The procedure works recursively and is similar to *refOneCluster* algorithm.

### 5.3 Empirical Evaluation

**Test Suite.** We ran multiple sets of experiments on problems of various sizes and with different value ranges for compatibility and activity parameters. The following is one snapshot across the entire topological problem space we explored, which we found,

**Algorithm 5.13.**

```

boolean AlternateActivation( $v, targetLevel; T_{cond}, \overline{T_{cond}}, V_{cond}$ ) {
  let vtCluster be an initially empty set of activations  $c : \langle cT, \overline{cT}, cV \rangle$ 
  visited( $v$ )  $\leftarrow$  true
  for each (inclusion activity constraint  $a \in \mathcal{C}_A$  with  $v$  as target variable)
    let aSources be an initially empty set of activations  $s : \langle sT, \overline{sT}, sV \rangle$ 
    aIsDiscarded  $\leftarrow$  false
    for each ( $a$ 's condition variable,  $u$ , s.t.  $u$  has not been visited and  $\lambda(u) \geq targetLevel$ ) {
      if (AlternateActivation( $u, targetLevel, sT, \overline{sT}, sV$ ) is false) {
        aIsDiscarded  $\leftarrow$  true; break
      }
      else {
        aSources  $\leftarrow$  aSources  $\cup$   $\langle sT, \overline{sT}, sV \rangle$ 
      } //end for  $u$ 
    } if (aIsDiscarded is false)
      if (aSources is empty)
        vtCluster  $\leftarrow$  vtCluster  $\cup$   $\langle T_{cond_a}, \overline{T_{cond_a}}, V_{cond_a} \rangle$ 
      else {
        aSources  $\leftarrow$  aSources  $\cup$   $\langle T_{cond_a}, \overline{T_{cond_a}}, V_{cond_a} \rangle$ 
        ANDActivity(aSources,  $cT, \overline{cT}, cV$ )
        vtCluster  $\leftarrow$  vtCluster  $\cup$   $\langle cT, \overline{cT}, cV \rangle$ 
      } //end else
    } //end for  $a$ 
  visited( $v$ )  $\leftarrow$  false
  if (vtCluster is empty)
    return false
  else {
    ORActivity(vtCluster,  $T_{cond}, \overline{T_{cond}}, V_{cond_v}$ )
    return true
  }
} //end AlternateActivation()

```

based on our results, to be representative for the performance comparison between solving a conditional CSP and the equivalent reformulated CSP.

The test suite has random conditional CSPs of 8 variables with domains of 6 values. The problems are organized in nine classes, each corresponding to a compatibility satisfiability,  $s_c$ , value in the  $[0.1 \dots 0.9]$  range. To avoid problems with very large solutions sets and thus to keep the average running time per problem under three minutes, we fixed the compatibility density at a low level,  $d_c = 0.15$ . The problem conditionality was given by  $d_a = s_a = 0.3$ . For each of the nine  $(d_c, s_c, d_a, s_a)$  problem classes we generated 100 problems.

Conditional CSPs were solved with *CCSP\_Mac* algorithm. Their null-based reformulations, obtained with the reformulation algorithm that handles activity cycles, are non-binary

standard CSPs. Therefore, these representations were transformed into binary constraint representations using the algorithm described in (Rossi, Petrie, & Dhar 1990). The binary null-based reformulations were solved with a standard maintaining arc consistency, *RefMAC*, algorithm. The two solving algorithms, *CCSP-Mac* and *RefMAC*, whose running times were compared, searched for all solutions. The comparison does not take into account the time to reformulate the conditional CSP into the non-binary null-based representation, nor the time to transform the non-binary representation into an equivalent binary representation.

**Results.** The execution time results are shown in Figure 5-xi, on a normal scale (left) and logscale (right). We observe that solving binary null-based reformulations is much slower, up to two orders of magnitude, than solving the original conditional CSP directly.

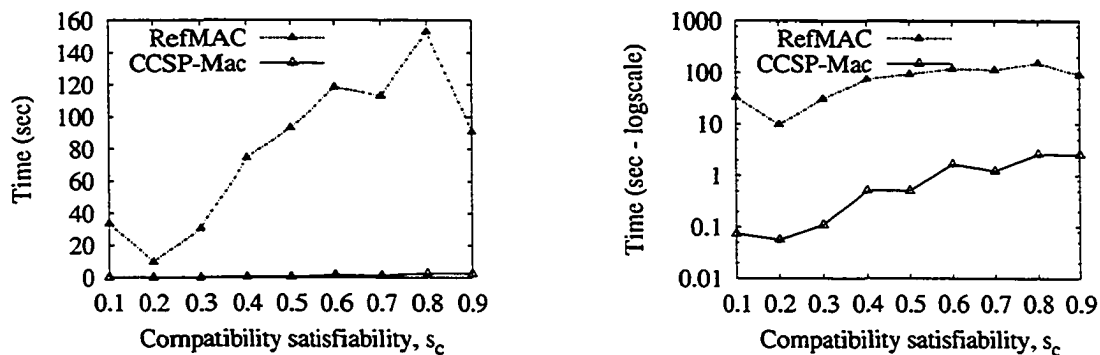


Figure 5-xi: Execution time of *CCSP-Mac*, which solves the conditional CSP, and *RefMAC*, which solves its binary null-based reformulation. The original conditional CSP has 8 variables and 6-value domains. 100 problem instances are generated in each topological class:  $s_c \in [0.1 \dots 0.9]$  and fixed  $d_c = 0.15$ ,  $d_a = s_a = 0.3$ .

There are two other measures that we use to compare the original and reformulated representation: domain size and solution size (Figure 5-xii). In the case of the conditional CSP, the domain size is fixed at 6. However, the binary null-based reformulations in the test suite have domains of variable size. This is the result of transforming the non-binary reformulations into binary representations.

It is known that this transformation is not practical due to increased spatial requirements

(Bacchus & van Beek 1998; Bessière 1999). The idea of transforming non-binary CSPs into equivalent binary CSPs is to generate new variables that represent (and replace) the non-binary constraints via the domain values of these variables. The values are the tuples of the non-binary constraints. New binary constraints are added between the new variable and all the variables on which the non-binary constraint is defined. Note that the arity of the non-binary constraints in the null-based reformulation determines the domain size of the new variables and the number of the additional binary constraints.

The Figure 5-xii shows domain size averages (left) and solution size averages (right) for the problems in the test suite described above.

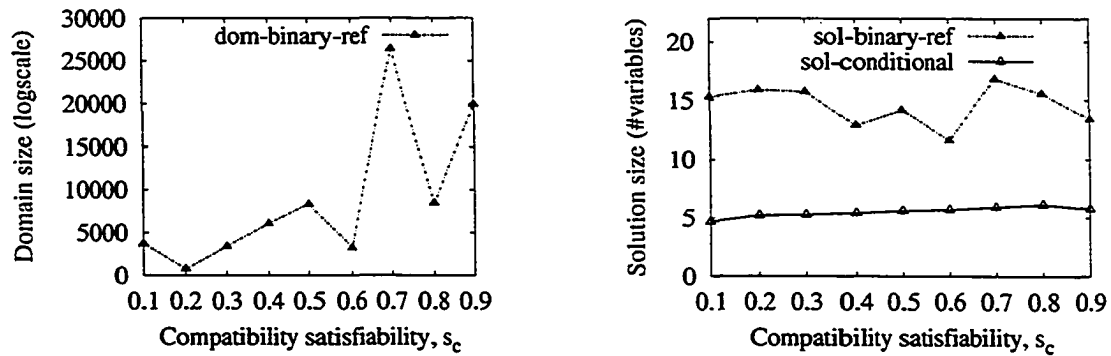


Figure 5-xii: Domain size averages (left) and solution size averages (right) of the binary null-based reformulations, dom-binary-ref and sol-binary-ref plots. These reformulations are obtained from the conditional CSPs in our test suite. The domain size of the conditional CSPs is 6 (not plotted). sol-conditional plot (right) shows the solution size averages for the original conditional CSPs.

The domain size average per 100 problem instances in a topological class in the case of the binary reformulation (see dom-binary-ref plot in Figure 5-xii left) is calculated by counting all values over all variables and by dividing the total by the number of variables. The solution size average per 100 problem instances in a topological class for both the binary reformulation and original conditional CSP (see sol-binary-ref and sol-conditional in Figure 5-xii right) is calculated by counting all variables participating in all solutions and by dividing the total by the number of solutions.

We observe that the domain size average reaches very large values, in the order of thousands and tens of thousands, and increases with the compatibility satisfiability parameter. The solution size average for the binary reformulation is 2 to 3 times larger than the solution size average for the conditional CSP.

The experimental results suggest that unless we find an efficient way to solve non-binary CSPs that reformulate conditional CSPs, directly solving the conditional CSP is significantly faster than solving the equivalent null-based reformulated CSP.

## 5.4 Summary

An alternative approach to direct solving methods is to reformulate a conditional CSP into its standard analog. The approach has the advantage of bringing to bear a mature constraint technology that has a wealth of advanced resolution algorithms. In this chapter we studied whether reformulation leads to solving methods that are faster than the direct methods. The contributions in this chapter are:

- developed an original formalism that transforms acyclic conditional CSPs into standard CSPs,
- designed a new reformulation algorithm that implements this formalism,
- derived an incremental reformulation under the assumption that problems do not have activity cycles,
- identified the activity cycle problem and designed a general null-based reformulation algorithm that deals with cycles,
- evaluated algorithm performance on CSP reformulations of random conditional CSPs, and compared it with the direct method performance.

Our experimental results showed that directly solving the conditional CSP outperforms solving the equivalent reformulated problem.

## CHAPTER 6

### CONCLUSION

Conditional constraint satisfaction problems are extensions to standard CSPs that have proved useful in representing configuration and diagnosis problems. In contrast with other CSP extensions, conditional CSP has not benefited from adaptations of efficient CSP solving algorithms to improve problem solution. Moreover, experimental analysis of the efficiency of available conditional CSP solvers has been extremely limited. Reformulating conditional CSPs into standard CSPs has been proposed in order to bring the full arsenal of CSP algorithms to bear. One reformulation approach adds null values to variable domains and transforms conditional CSP constraints into CSP constraints. However, a complete null-based reformulation of conditional CSPs has not been available.

In this thesis we researched more efficient solvers for conditional CSPs. The research findings are the result of examining three topics:

1. Advanced algorithms for solving conditional CSPs,
2. Thorough empirical evaluation of the proposed solving methods, and
3. New reformulation algorithms and experimental analysis of their efficiency.

In the following we summarize the contributions produced for each topic.

We designed and developed two advanced algorithms for conditional CSP that adapt local consistency methods of forward checking and maintaining arc consistency to conditional CSP solving. The technical challenges encountered and overcome in designing these algorithms were:

- to monitor the activity status of problem variables as determined by consistency checking of activity constraints,

- to enforce the chosen level of consistency when checking both compatibility and activity constraints,
- in the case of maintaining arc consistency, to extend arc consistency with activation consistency along activity constraints.

The opportunity of importing efficient standard algorithms, whose behavior has been extensively tested in the standard domain, has raised new challenges with regard to conducting similar testing in the conditional domain. The reality of many applications that use the conditional CSP framework is that either real-life problem data is not publicly available, or problem examples are too simple. A practical approach adopted in this thesis, which has proved very successful for benchmarking standard solving algorithms, was to use randomly generated conditional CSPs. To evaluate empirically the proposed algorithms:

- large and diverse problem populations were generated using a conditional CSP random generation model (Wallace 1996),
- relative performance of the new algorithms and a modified backtrack search version was measured by:
  - comparing running times when algorithms produced solutions of minimum size,
  - counting search operations, such as number of backtracks, compatibility checks, and condition checks, when algorithms found all solutions.

The testing showed that the performance of the advanced algorithms is up to two orders of magnitude better than plain backtrack search.

The reformulation studied in this thesis is based on adding a “null” value to the domains of those variables whose activity is conditioned by satisfying activity constraints during search. A null-based reformulation of conditional CSPs was presented and studied in depth in (Gelle 1998; Soininen, Gelle, & Niemelä 1999). However, that transformation is limited in the following key respects:

- It does not transform multiple activations of the same variable,

- It does not preserve locality of change, i.e., the reformulated problem cannot be updated locally when the original problem changes with the additional of another activity constraint to a multiple activation cluster, and
- It does not handle activity cycles.

We addressed these limitations of the null-based reformulation by developing two alternative transformations. One removes the first two limitations; the other removes the third. Both algorithms synthesize non-binary ordinary constraints whose arity increases with the number of activity constraints in a multiple activation cluster or in an activity cycle. The second algorithm, which does not impose any restriction on the structure of the original conditional CSP, was used to evaluate experimentally whether reformulation offers a more efficient solution. The testing showed that

- greater efficiency in solving conditional CSPs lies with algorithms that operate directly on the original representation,
- much has to be learned about what is specific to null-based reformulations and how standard methods can more efficiently exploit these representations.

We envision three directions for our future work. In (Gelle 1998; Gelle & Faltings 2003) a different reformulation method has been proposed that generates a set  $S$  of standard CSPs equivalent to the original conditional CSP. Conventional local consistency methods are then applied on intermediate problems generated along the way to producing  $S$  in order to reduce the size of  $S$  and solve its members more efficiently with standard CSP solving algorithms. Gelle's reformulation is a general formalism that handles a more general class of conditional CSP, which contain mixed constraints that involve both discrete and numeric variables. Its implementation together with local-consistency and standard search algorithms is in Common Lisp and Maple. The experimental analysis uses several real world problems from configuration and design which exhibit mixed constraints.

A very interesting fact is that Gelle's CSP-generation reformulation, which is based on processing activity constraints one at a time, has to consider a certain ordering of the

activity constraints. This ordering ensures that an activity constraint is applied iff its condition variables are already active as a result of having previously processed activity constraints that target these condition variables. The ordering is possible if there is no cyclic dependency among activity constraints. Otherwise, cyclic activity constraints have to be detected and processed as a group rather than individually.

We are interested in extending the evaluation of our algorithms by testing their performance against a solver that uses Gelle's CSP-generation reformulation and applies standard methods to the resulting CSP set. The first step is to develop a C++ implementation of Gelle's reformulation and integrate it in the software system that we used in our experiments. This system has an object-oriented infrastructure that integrates the implementation of all the local-consistency and solving algorithms for both standard and conditional CSP, as well as the null-based reformulation algorithms. The new solver will be thoroughly tested using random conditional CSPs. New metrics will be determined since the reformulation algorithm of the new solver has a preprocessing phase that (1) transforms exclusion activity constraints into compatibility constraints, (2) creates an inclusion activity graph, (3) transforms the graph such that cycles are eliminated, and (4) orders the inclusion activity constraints.

Another research direction of interest is to find better algorithms for solving null-based reformulations. In general, a conditional CSP has extremely large solutions spaces. However, they are partitioned into sets that share the same variables (active variables which are assigned values). In the null-based reformulation, these sets are extended with null values for all the other problem variables (which do not participate in the solutions to the original conditional CSP problem). These similarities among solution subspaces in a null-based reformulated problem suggest that precompiling the null-based representation to condense its solution spaces is worth pursuing. This approach is based on the idea of interchangeability (Freuder 1991). More exactly, we are interested in the application of the structuring algorithm that produces CSP precompilations using meta interchangeability (Weigel 1998).

The third direction for future work is motivated by the fact that real-life configura-

tion and diagnosis problems are formulated as non-binary conditional CSPs. We want to generalize the current implementations of the advanced direct solving methods to handle non-binary constraints and take advantage of efficient non-binary local consistency algorithms (Bessière & Régin 1997; 2001; Bessière *et al.* 2002).

## REFERENCES

- Achlioptas, D.; Kirousis, L.; Kranakis, E.; Krizanc, D.; Malloy, M.; and Stamatiou, Y. 2001. Random constraint satisfaction: A more accurate picture. *Constraints* 6(4):329–344.
- Bacchus, F., and van Beek, P. 1998. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15 National Conference Artificial Intelligence (AAAI'98)*, 311–318.
- Bessière, C., and Regin, J.-C. 1996. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In Freuder, E., ed., *Principles and Practice of Constraint Programming. Lecture Notes of Computer Science*, volume 1118, 61–75. Springer. (CP'96: Second International Conference, Boston, MA, USA).
- Bessière, C., and Régin, J.-C. 1997. Arc consistency for general constraint networks: preliminary results. In *Proceedings the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, 398–404.
- Bessière, C., and Régin, J.-C. 2001. Refining the basic constraint propagation algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, 309–315.
- Bessière, C.; Meseguer, P.; Freuder, E.; and Larrosa, J. 2002. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence* 141:205–224.
- Bessière, C. 1991. Arc-consistency in dynamic constraint satisfaction problems. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, 221–226.
- Bessière, C. 1999. Non-binary constraints. In *Proceedings of the Principles and Practice of Constraint Programming (CP'99)*.
- Dechter, R., and Dechter, A. 1988. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88*, 37–42.
- El Fattah, Y., and Dechter, R. 1992. Empirical evaluation of diagnosis as optimization in constraint networks. In *Working Notes of the Third International Workshop on Principles of Diagnosis (DX-92)*.
- Faltings, B.; Freuder, E.; Friedrich, G.; and Felfernig, A., eds. 1999. *Configuration - Papers from the AAAI Workshop*. AAAI Press. Technical Report WS-99-05.
- Frayman, F., and Mittal, S. 1987. Cosask: A constraint-based expert system for configuration tasks. In Sriram, D., and Adey, R., eds., *Knowledge-Based Expert Systems in Engineering: Planning and Design*. Computational Mechanics Publications. 143–166.
- Freuder, E., and Mackworth, A., eds. 1994. *Constraint-Based Reasoning (Special Issue of Artificial Intelligence: An International Journal)*. MIT Press.
- Freuder, E., and Wallace, R. 1992. Partial constraint satisfaction. *Artificial Intelligence* 58.
- Freuder, E. 1978. Synthesizing constraint expressions. *Communications of ACM* 21(11):958–966.

- Freuder, E. 1991. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, 227–233.
- G., D., and Kaiser, T. 1997. Determining the availability of distributed applications. In Lazar, A.; Saracco, R.; and Stadler, R., eds., *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*, 207–218. Chapman & Hall.
- Gaschnig, J. 1974. A constraint satisfaction method for inference making. In *Proceedings of the Twelfth Annual Allerton Conference on Circuit and System Theory*.
- Gelle, E., and Faltings, B. 2003. Solving mixed and conditional constraint satisfaction problems. *Constraints* 8(2):107–141.
- Gelle, E. 1998. *On the generation of locally consistent solution spaces in mixed dynamic constraint problems*. Ph.D. Dissertation, Departement d'Informatique, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland.
- Gent, I., and Walsh, T. 1999. *CSplib: a benchmark library for constraints*. <http://csplib.cs.strath.ac.uk>.
- Grant, S., and Smith, B. 1995. The phase transition behavior of maintaining arc consistency. Technical Report 92.95, School of Computing, University of Leeds. (A revised and shortened version appears in *Proceedings ECAI'96*, pp. 175–179, 1996).
- Hamscher, W. and Console, L., and de Kleer, J., eds. 1992. *Readings in Model-Based Diagnosis*. San Mateo, CA: Morgan Kaufmann Publishers.
- Haralick, R., and Elliott, G. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14:263–313.
- Haselböck, A. 1993. *Knowledge-based configuration and advanced constraint technologies*. Ph.D. Dissertation, Institut für Informationssysteme, Technische Universität Wien, Vienna, Austria.
- Houck, K.; Calo, S.; and Finkel, A. 1995. Towards a practical alarm correlation system. In Sethi, A.; Raynaud, Y.; and Faure-Vincent, F., eds., *Proceedings of the Fourth IFIP/IEEE International Symposium on Integrated Network Management*, 226–237. Chapman & Hall.
- Huard, S., and Freuder, E. 1993. A debugging assistant for incompletely specified constraint network knowledge bases. *International Journal of Expert Systems: Research and Applications* 419–446.
- Katker, S., and Paterok, M. 1997. Fault isolation and even correlation for integrated fault management. In *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*.
- Keirouz, W.; Kramer, G.; and Pabon, J. 1995. Exploiting constraint dependency information for debugging and explanation. In *Principles and Practice of Constraint Programming*, 183–196. Cambridge, MA: The MIT Press.
- MacIntyre, E.; Prosser, P.; Smith, B.; and Walsh, T. 1998. Random constraint satisfaction: Theory meets practice. In *Principles and Practice of Constraint Programming (CP-98)*, 325–339. Springer Verlag.
- Mackworth, A. 1977. Consistency in networks of relations. *Artificial Intelligence*.

- Mailharro, D., ed. 2003. *IJCAI 2003 Configuration Workshop*. In conjunction with the 18th International Joint Conference on Artificial Intelligence. <http://www2.ilog.com/ijcai-03/>.
- Mittal, S., and Davis, H. 1989. Representing and solving hierarchical constraint problems. Technical report, Xerox PARC.
- Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*.
- Mittal, S., and Frayman, F. 1989. Towards a generic model of configuration tasks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1395–1401. Morgan Kaufmann Publishers, Inc.
- Riese, M. 1993. *Model-based diagnosis of Communication Protocols*. Ph.D. Dissertation, Swiss Federal Institute of Technology, Lausanne.
- Rose, M. 1993. Challenges in network management. *IEEE Network* 7(6):16–19.
- Rossi, F.; Petrie, C.; and Dhar, V. 1990. On the equivalence of constraint satisfaction problems. In *In the Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90)*, 550–556.
- Sabin, D., and Freuder, E. 1994. Contradicting conventional wisdom in constraint satisfaction. In Borning, A., ed., *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science*, volume 874. Springer. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).
- Sabin, D., and Freuder, E. 1996. Composite constraint satisfaction for configuration. In Freuder, E., ed., *Working Notes of the AAAI'96 Fall Symposium*.
- Sabin, D.; Sabin, M.; Russell, R.; and Freuder, E. 1995. A constraint-based approach to diagnosing software problems in computer networks. In Montanari, U., ed., *Principles and Practice of Constraint Programming - CP'95, Lecture Notes of Computer Science 976*. Springer Verlag.
- Sabin, M., and Freuder, E. 1996. Automated construction of constraint-based diagnosticians. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis (DX'96)*.
- Sabin, M., and Freuder, E. 1998. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. In *Web-published papers of the CP'98 Workshop on Constraint Problem Reformulation*.
- Sabin, M., and Freuder, E. 1999. Detecting and resolving inconsistency in conditional constraint satisfaction problems. *Proceedings of the AAAI'99 Workshop on Configuration* 95–100.
- Sabin, M.; Bakman, A.; Freuder, E.; and Russell, R. 1999. A constraint-based approach to fault management for groupware services. In *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*.
- Sabin, M.; Russell, R.; and Freuder, E. 1997. Generating diagnostic tools for network fault management. In *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*.

- Sabin, M.; Russell, R.; and Miftode, I. 2001. Using constraint technology to diagnose errors in networks managed with spectrum. In *Proceedings of the IEEE International Conference on Telecommunications*.
- Soininen, T.; Tiihonen, J.; Männistö, T.; and Sulonen, R. 1998. Towards a general ontology of configuration. *Artificial Intelligence in Engineering, Design, and Manufacturing* 12:357–372.
- Soininen, T.; Gelle, E.; and Niemelä, I. 1999. A fixpoint definition of dynamic constraint satisfaction. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*.
- Sqalli, M. and Freuder, E. 1998. Integration of CSP and CBR to compensate for incompleteness and incorrectness of models. In *AAAI-98 Spring Symposium on Multimodal Reasoning*.
- Tsang, E. 1993. *Foundations of Constraint Satisfaction*. London: Academic Press Limited.
- Verfaillie, G., and Schiex, T. 1994. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the 12th AAAI*, 307–312.
- Wallace, R. 1996. *Random CSP Generator*. Constraint Computation Center, University of New Hampshire, Durham, NH, U.S.A. <http://www.cs.unh.edu/ccg/code.html>.
- Weigel, R. 1998. *Abstractions and Reformulations in Dynamic Constraint Satisfaction*. Ph.D. Dissertation, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.