View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Saskatchewan's Research Archive

# Empirical Evaluation of Soft Arc Consistency Algorithms for Solving Constraint Optimization Problems

A Thesis Submitted to the

College of Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Xiaonuo Gantan

©Xiaonuo Gantan, August, 2011. All rights reserved.

# Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C9

# Abstract

A large number of problems in Artificial Intelligence and other areas of science can be viewed as special cases of constraint satisfaction or optimization problems. Various approaches have been widely studied, including search, propagation, and heuristics. There are still challenging real-world COPs that cannot be solved using current methods.

We implemented and compared several consistency propagation algorithms, which include W-AC*2001 (Cooper and Schiex, 2004), EDAC (Givry and Zytnicki, 2005), VAC (Cooper *et al.*, 2010), and xAC (Horsch *et al.*, 2002). Consistency propagation is a classical method to reduce the search space in CSPs, and has been adapted to COPs. We compared several consistency propagation algorithms, based on the resemblance between the optimal value ordering and the approximate value ordering generated by them. The results showed that xAC generated value orderings of higher quality than W-AC*2001 and EDAC.

We evaluated some novel hybrid methods for solving COPs. Hybrid methods combine consistency propagation and search in order to reach a good solution as soon as possible and prune the search space as much as possible. We showed that the hybrid method which combines the variant TP+OnOff (Section 3.3) and branch-and-bound search (Section 3.5) performed fewer constraint checks and searched fewer nodes than others in solving random and real-world COPs.

# Acknowledgements

There are many people who deserve my sincere gratefulness. In particular, I want to thank Dr. Michael C. Horsch for his excellent supervision, patient guidance, unconditional support, and sparking inspiration throughout my graduate studies. Many thanks are due to the other supervisory committee members: Dr. Anthony J. Kusalik and Dr. Kevin Stanley.

This is the thesis is dedicated to my father, who set up an example for me on how to be consistent and devoted to my work. It is also dedicated to my mother, who loved me unconditionally and supported me no matter how hard the graduate studies are.

# Contents

**Permission to Use** i

**Abstract** ii

**Acknowledgements** iii

**Contents** v

**List of Tables** vii

**List of Figures** viii

**List of Abbreviations** ix

**1 Introduction** 1

**2 Literature Review** 5
- 2.1 Constraint Satisfaction Framework . . . 5
- 2.2 Algorithms for CSPs . . . 7
  - 2.2.1 Tree Search . . . 7
  - 2.2.2 Consistency Propagation . . . 10
  - 2.2.3 Heuristics . . . 13
  - 2.2.4 Local Search . . . 14
  - 2.2.5 Summary of CSP Algorithms . . . 15
- 2.3 Constraint Optimization Problems . . . 15
  - 2.3.1 Hard and Soft Constraints . . . 16
  - 2.3.2 Valued Constraint Satisfaction Problems . . . 16
  - 2.3.3 Semiring-Based Constraint Satisfaction Problems . . . 18
  - 2.3.4 Comparison of SCSPs and VCSPs . . . 23
  - 2.3.5 Weighted CSPs . . . 25
- 2.4 Summary . . . 26

**3 Soft AC Algorithms** 27
- 3.1 Preliminary . . . 27
  - 3.1.1 Conventions . . . 27
  - 3.1.2 Arc Consistency Closure . . . 27
- 3.2 W-AC*2001 and EDAC . . . 28
  - 3.2.1 Foundation . . . 28
  - 3.2.2 W-AC*2001 and Directional Arc Consistency . . . 30
  - 3.2.3 Enforcing Arc Consistency . . . 34
  - 3.2.4 EDAC . . . 38
- 3.3 xAC . . . 40
- 3.4 Virtual Arc Consistency . . . 46
  - 3.4.1 Propagating VAC . . . 47
- 3.5 Search . . . 48
- 3.6 Summary . . . 49

**4 Comparing the Value Ordering Heuristics of xAC, W-AC*2001, and EDAC** 51
- 4.1 Purpose . . . 51
- 4.2 Problem Instances . . . 52

4.3 Methods . . . 53
4.4 Empirical Results . . . 54

**5 Empirical Comparison of Soft AC Algorithms in Solving Random COPs 56**
5.1 Preliminary . . . 56
5.2 MaxCSPs . . . 57
5.3 Uniform Integer COPs . . . 59
5.4 Summary . . . 61

**6 Empirical Comparison of xAC, EDAC and VAC in Solving Real-World COPs 63**
6.1 Uncapacitated Warehouse Location Problems . . . 63
6.1.1 WCSP Formulation of UWLP . . . 64
6.1.2 Examples . . . 65
6.1.3 Empirical Results . . . 68
6.2 Radio Link Frequency Assignment Problems . . . 69
6.2.1 Informal Description . . . 70
6.2.2 Formal Definition of the CELAR Problems . . . 71
6.2.3 WCSP Formulation of RLFAP . . . 73
6.2.4 Experimental Results . . . 74
6.3 Quasigroup Problems . . . 76
6.3.1 Problem Description . . . 76
6.3.2 Problem Generations and Encodings . . . 77
6.3.3 Experimental Results . . . 77
6.4 Conclusion . . . 83

**7 Conclusion and Future Work 84**
7.1 Conclusions and Contributions . . . 84
7.2 Future Work . . . 85

**References 87**

**A RLFAP Data Files 90**

**DEFINITION INDEX 92**

# List of Tables

6.1 Storage Costs . . . . . . . . . . 65
6.2 Shipment Cost . . . . . . . . . . 66
6.3 Unary Constraints for Warehouses . . . . . . . . . . 66
6.4 Unary Constraints for Stores . . . . . . . . . . 66
6.5 UWLP instances . . . . . . . . . . 67
6.6 Number of Constraint Checks on UWLPs . . . . . . . . . . 68
6.7 RLFAP Instances . . . . . . . . . . 74
6.8 Number of Constraint Checks for Solving RLFAPs . . . . . . . . . . 75
6.9 Number of Nodes Searched for Solving RLFAPs . . . . . . . . . . 75
6.10 Quasigroup Examples . . . . . . . . . . 76

# List of Figures


2.1 An example of search tree including two variables . . . . . 8
2.2 Structure of a Constraint . . . . . 21
2.3 An Example of Combination and Projection . . . . . 22
2.4 From SCSP to VCSP . . . . . 24
2.5 From VCSP to SCSP . . . . . 25

3.1 A WCSP instance that is not $NC*$ . . . . . 32
3.2 A WCSP instance that is $NC*$ . . . . . 33
3.3 A WCSP instance that is $DAC*$ . . . . . 33
3.4 A WCSP instance that is $AC^*$ but not $DAC^*$ . . . . . 34
3.5 A WCSP instance that is $FDAC^*$ but not $EDAC^*$ . . . . . 39
3.6 A WCSP instance that is $EDAC^*$ . . . . . 40
3.7 The parameters of an arbitrary node X . . . . . 43

4.1 Two Binary Constraints in A Skewed COP Instance . . . . . 52
4.2 Quality Comparison of Value Ordering Heuristics . . . . . 55

5.1 Comparing Number of Constraint Checks on MaxCSPs . . . . . 57
5.2 Comparing Number of Nodes on MaxCSPs . . . . . 58
5.3 Comparing Number of Constraint Checks on UICOPs . . . . . 60
5.4 Comparing Number of Nodes on UICOPs . . . . . 60

6.1 Binary Constraints . . . . . 67
6.2 Number of Constraint Checks Performed for Solving Quasigroups . . . . . 80
6.3 Number of Nodes Searched for Solving Quasigroups . . . . . 82

# List of Abbreviations

| | |
|---|---|
| AC | Arc Consistency |
| ACS | Arc Consistency during Search |
| BJ | Backjumping |
| BM | Backmarking |
| BnB | Branch and Bound |
| BT | Backtracking |
| CARD | Minimum Cardinality |
| CP | Consistency Propagation |
| CS | Constraint System |
| CSP | Constraint Satisfaction Problems |
| COP | Constraint Optimization Problems |
| DAC | Directional Arc Consistency |
| EAC | Existential Arc Consistency |
| EDAC | Existential Directional Arc Consistency |
| FDAC | Full Directional Arc Consistency |
| FEAS | Feasibility |
| GI | Generalized Interval |
| GT | Generate and Test |
| MAX | Maximum Feasibility |
| MaxCSP | Max Constraint Satisfaction Problems |
| MSAC | Maintaining Soft Arc Consistency |
| NC | Node Consistency |
| NP | Non-deterministic Polynomial-time |
| PCC | Pearson's Correlation Coefficient |
| QCP | Quasigroup Completion Problem |
| QWH | Quasigroup With Holes |
| RLFAP | Radio Link Frequency Assignment Problem |
| RP | Radical Pruning |
| SAC | Soft Arc Consistency |
| SCSP | Semiring-based Constraint Satisfaction Problems |
| SPAN | Minimum Span |
| SRCC | Spearman's Rank Correlation Coefficient |
| TP | Tree Pruning |
| UICOP | Uniform Integer Constraint Optimization Problem |
| UWLP | Uncapacitated Warehouse Location Problem |
| VAC | Virtual Arc Consistency |
| VCSP | Valued Constraint Satisfaction Problems |
| W-AC*2001 | Weighted Arc Consistency 2001 |
| WCSP | Weighted Constraint Satisfaction Problems |

# Chapter 1

# Introduction

Constraint satisfaction problems (CSPs) and constraint optimization problems (COPs) include many real-world applications in machine vision, belief maintenance, scheduling, and others. Because of the various applications in which CSPs and COPs are useful, extensive research has been devoted into developing more efficient algorithms for solving them. After decades of hard work by researchers, the understanding of CSPs and the development of algorithms have brought benefits to the real world. The practical application of COPs, however, still have lots of questions waiting to be answered. Based on one of the popular COP frameworks, this thesis concentrates on improving the performance of the xAC algorithm (Horsch *et al.*, 2002), and aims at providing comprehensive empirical results by comparing different state-of-the-art algorithms based on their empirical performance.

A CSP includes a set of variables, a finite and discrete domain for each variable, and a set of constraints. Each constraint is a subset of the Cartesian product of all the domains of some variables. The goal is to find an assignment of values to all the variables such that this assignment satisfies all the constraints. This assignment is called a solution to the problem. On the other hand, if there is no such assignment, the problem has no solution. Typically, we stop when we find a solution, but we may want to find all solutions for some problems.

Constraints are classified as unary, binary, or n-ary constraints, based on the number of variables involved. A unary constraint includes only one variable, and prevents some of the values in the domain from being assigned to the variable. Similarly, a binary constraint includes two variables, and prohibits some illegal assignments of value combinations. The COP instances in this thesis contain only unary or binary constraints, because any $k$-ary ($k > 2$) constraints can be converted

to several binary constraints.

The following is an example of constraint satisfaction problem. Establishing a network of radio links gives rise to the Radio Link Frequency Assignment Problem (RLFAP) (see Page 70). A radio link is a communication channel between a pair of radio transmitters. A signal can be transmitted through a radio link from one transmitter to the other. Each radio link must be assigned an operating frequency from a set of available frequencies. The distance between links is measured by the difference of the frequencies assigned to them. The assignment complies with certain preferences, regulations, and physical locations of the transmitters. First, some links may have preassigned frequencies. Second, when two transmitters each of which hosts a different link are physically close to each other, the difference of these two frequencies have to be large enough so the communication is not distorted. Third, each link has a reverse link. A reverse link allows simultaneous transmission between transmitters. The frequencies assigned to each reverse link have to differ from the original link by a certain distance. A link and its reverse link occur in pairs.

A radio link frequency assignment problem can be modelled as a CSP by declaring each radio link as a variable (Cabon *et al.*, 1999; Freuder and Wallace, 1992). The domain of each variable is the set of available frequencies for that link. Unary constraints specify preassigned frequencies. Binary constraints specify that the frequencies assigned to any two links should be different from each other by a certain amount. To solve an RLFAP is to find an assignment of all the links that comply with all the constraints.

Based on the current constraint satisfaction framework, an RLFAP can be solved by various methods, including tree search, local search, consistency propagation, and heuristics (see Chapter Two). No matter which method is chosen, the constraint satisfaction framework allows certain flexibility to deal with different problem instances. For example, with some possible minor modifications, altering the preassigned frequencies and the smallest distance between each pair of interfering link assignments will not require changing the solving technique, although the solution of the problem may change.

Although CSPs can be used to model a large proportion of real world examples, sometimes a

CSP can be over-constrained where no solution exists. For example, in an RLFAP, if the originally assigned frequency of the link in one direction has to differ from the reverse one by a large distance, there may be no assignment satisfying all constraints.

In order to effectively solve over-constrained CSPs, each constraint is associated with violation costs (Schiex *et al.*, 1995; Bistarelli *et al.*, 1999). These associations transform CSPs into COPs, which can be solved by finding an assignment that minimizes the total costs of all violated constraints. The costs or weights indicate the preference or importance over different constraints, thus the assignment minimizing the costs is the most preferable one. These costs or weights are part of problem descriptions and are assessed by domain experts. Considering a radio link frequency problem, if the hard constraints between some links cannot be broken under all conditions, while the soft constraints between other links can be broken for a certain cost, then the final solution satisfies all the hard constraints and violates as few soft constraints as possible.

Constraint optimization algorithms are built on top of constraint satisfaction techniques, and constraint optimization frameworks extend constraint satisfaction models. Because of the valuation of each partial assignment and the comparison between different assignments, the computational complexity of COPs is higher than CSPs (Cohen *et al.*, 2006). Since the time complexity of COPs is nondeterministic polynomial, more efficient algorithms are needed. Most of the current COP algorithms are developed by extending their counterparts in solving CSPs. The popular options include branch-and-bound Search, Russian Doll Search (Verfaillie *et al.*, 1996), Partial Consistency Propagation (Verfaillie *et al.*, 1996), and Soft Node and Arc Consistency (Larrosa, 2002; Cooper and Schiex, 2004; Cooper *et al.*, 2010).

In this thesis, the major part of the work is devoted to developing more efficient heuristic inference algorithms which generate value orderings to guide the branch-and-bound search, and comparing our heuristic algorithms against various state-of-the-art consistency propagation algorithms. The algorithm we improved is xAC (Horsch *et al.*, 2002). Since xAC and other algorithms only generate value ordering and perform consistency propagation without any search, we combined these algorithms with a plain branch-and-bound search and compared the performance of

the hybrid algorithms. The branch-and-bound algorithm is introduced by Land and Doig (1960).

A summary of this thesis is:

- Study of consistency propagation (CP) and value ordering heuristics in a COP framework;
- Implementation of W-AC*2001, EDAC, VAC, xAC and variable ordering heuristics combined with a branch-and-bound search algorithm;
- Proposal of several xAC variations (Section 3.3) to guide the search;
- Extensive empirical comparisons of different hybrid algorithms for both random and real-world COP solving;
- Analysis of the performance of using different parameters for the xAC algorithm.

This is the first work that systematically studies the value ordering heuristic provided by xAC and different xAC variations in a branch-and-bound search for solving COPs. It is also the most extensive work to explore the potential of the xAC algorithm.

The rest of this thesis is structured as follows. Chapter 2 gives a literature review of the background and the related work on CSP/COP frameworks and the algorithms. Chapter 3 introduces a group of so-called "soft" AC properties (Cooper and Schiex, 2004) and propagation algorithms. Chapter 4 gives a comparison of the value ordering heuristics generated by xAC, W-AC*2001, and EDAC. Chapter 5 shows an empirical comparison of xAC, EDAC, and VAC on solving random COPs. Chapter 6 compares the xAC, EDAC, and VAC on solving several real world COPs.

# Chapter 2

# Literature Review

This chapter gives a brief description of the frameworks for modelling CSPs and COPs and relevant arc consistency algorithms. Section 2.1 introduces the classical CSP framework. Section 2.2 introduces classical CSP algorithms. Section 2.3 introduces two popular COP frameworks, valued CSPs (Schiex *et al.*, 1995) and semiring-based CSPs (Bistarelli *et al.*, 1999), which are used to model weighted CSPs.

## 2.1 Constraint Satisfaction Framework

**Definition 2.1.** *Constraint Satisfaction Problem (CSP). A Constraint Satisfaction Problem is a tuple* $P = (X, D, C)$*, where*

- $X = \{x_1, x_2, \cdots, x_n\}$ *is a set of variables;*
- $D = \{d_1, d_2, \cdots, d_n\}$ *is a set of finite domains, each* $d_i$ *contains values* $\{v_1, v_2, \cdots, v_k\}$ *that can be assigned to variable* $x_i$*, where* $k$ *is an integer that may be different for different* $i$*;*
- $C$ *is a set of constraints, each of which allows some simultaneous assignments of some values to certain variables and forbids the rest. Each constraint is a subset of the total cartesian product of all the domains of involved variables: for constraint* $c_i$ *defined over variables* $x_{i_1}, x_{i_2}, \cdots, x_{i_j}$, $c_i \subset d_{i_1} \times d_{i_2} \times \cdots \times d_{i_j}$.

Each variable in $X$ has a corresponding domain in $D$, which contains a set of distinct values. Any of these values can be assigned to this variable.

An example of CSP is scheduling a birthday party time. Each guest has his or her own schedules. Some of them may work night shifts, while others work during the day. What is the best time so

that everyone can join the party? In this case, the set of variables would be $X = \{x_1, \cdots, x_n\}$ where $x_i$ is a potential party guest. The set of domains would be $D = \{d_1, \cdots, d_n\}$ where $d_i$ is a set of time slots when $x_i$ is available. The set of constraints would be $C = \{c_1, c_2, \cdots, c_n\}$, where $c_i$ forbids certain time slots being assigned to $x_i$ because the guest $i$ will not be available for those time slots. Such a small question can be answered by manually selecting a time slot when all guests are available. However, if it is a shareholders' meeting about a company's annual financial report, it might be hard to find the perfect time slot so that all significant shareholders can attend the meeting. When the number of people involved becomes bigger and bigger, a good algorithm for solving general CSPs will save a lot of time and effort.

The following overview is based on binary CSPs, in which $C$ only contains unary and binary constraints. A unary constraint involves only one variable and a binary constraint involves two. Similarly a $N$-ary constraint prohibits certain value assignments between $N$ variables. Since N-ary ($N > 2$) constraints can be transformed into binary constraints, we restrict our attention to binary constraints without loss of generality (Bacchus and Beek, 1998) .

The goal of solving CSPs is to assign a value $v$ to each variable $x$ such that the whole assignment satisfies all the constraints. Such an assignment is called a solution. All the possible assignments form the Cartesian product of all the domains: $D = d_1 \times d_2 \times \cdots \times d_n$. Because the size of the Cartesian product grows exponentially with the number of variables, a brute force algorithm searching all possibilities to find the optimal answer is impractical and we need intelligent algorithms to prune unnecessary search.

Several necessary definitions to formalize CSPs are defined as follows (Dechter, 1990).

**Definition 2.2.** *Constraint . A Constraint $c_i$ is a relation $r_i$ defined on a subset of variables $s_i \subseteq X$. The relation denotes the variables' simultaneous legal value assignments. $s_i$ is called the scope of $r_i$.*

If $s_i = \{x_{i_1}, \cdots, x_{i_r}\}$, then $r_i$ is a subset of the Cartesian product $d_{i_1} \times \cdots \times d_{i_r}$. Thus, a constraint can also be viewed as a pair $c_i = \langle s_i, r_i \rangle$.

**Definition 2.3.** *Assignments . For a $CSP = (X, D, C)$, a single assignment of $x_i$ is an assignment*

*of a value $v \in d_i$ to $x_i$; a partial assignment of a set of variables $Y \in X$ is a set of single assignments for each variable in $Y$; a full assignment is a partial assignment whose set of variables is equal to $X$. A partial assignment is consistent if it satisfies all of the constraints whose variables are assigned.*

**Definition 2.4.** *Solution . A solution is a full assignment such that $\forall c \in C$, $c$ is satisfied by a partial assignment whose set of variables $Y$ is a subset of $X$.*

## 2.2 Algorithms for CSPs

Three categories of algorithms have become the standard techniques for solving CSPs: *tree search*, *consistency propagation*, and *local search*.

### 2.2.1 Tree Search

For any CSP, suppose we have an ordering of the variables $x_1, x_2, \cdots, x_n$, following which we assign a value to each of the variables. Then we visualize the search as a tree. The root node is $\{\}$. The remaining nodes in the tree represent single assignments and the edges connect different single assignments. A path from the root to any node is a partial assignment. For example, Figure 2.1 shows a full search tree for two variables, in which $x_1$ has two values, and $x_2$ has two values. A solution will be a path from the root down to one of the leaves whose values satisfy all constraints in this problem.

#### Generate and Test

Generate and Test (GT) generates an assignment in which each variable is assigned a value, and tests whether the assignment satisfies all of the constraints. If not, GT tries another assignment. Although GT can be implemented as a Depth First Search, it still has to traverse the whole search space in the worst case. GT is not efficient because the worst case time complexity is exponential in the number of variables. Therefore, GT is impractical except for very small problems.

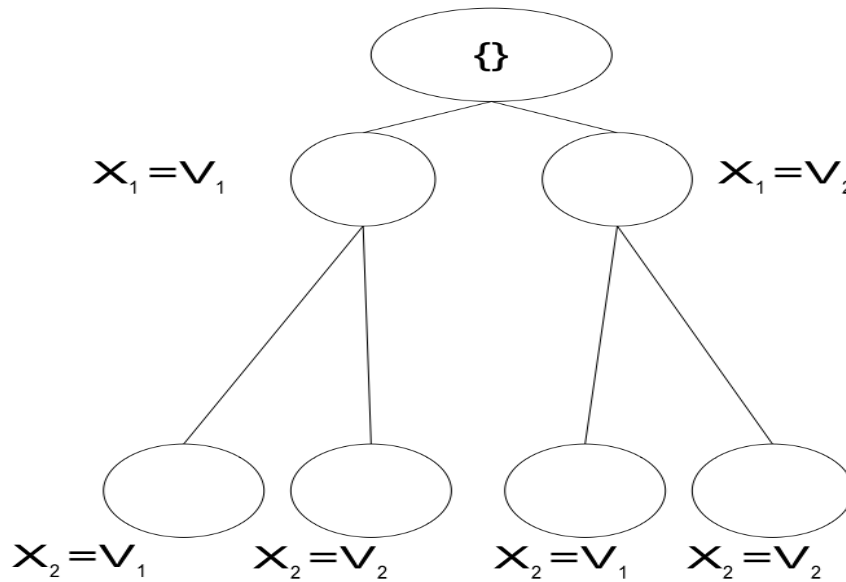



**Figure 2.1:** An example of search tree including two variables

**Backtracking**

Instead of traversing the whole search space (i.e., the Cartesian product of all the variable domains), Backtracking (BT) prunes inconsistent partial assignments. Some partial assignments are inconsistent because they violate some constraints. If a partial assignment is inconsistent, it cannot be extended to a consistent assignment by assigning more values to the uninstantiated variables. BT uses this property to stop assigning values to variables following an inconsistent assignment. If there are more values to try in the current variable, BT tries to assign those values in some order. Otherwise, BT backtracks to the most recently assigned variable.

Because of thrashing, which means *search in different parts of the space keeps failing for the same reasons* (Kumar, 1992), BT's performance is still exponential in the number of variables. One cause leading to thrashing is *node inconsistency*, in which a certain variable's values do not satisfy a unary constraint on that variable, thus leading to an inconsistent assignment each time

BT tries to assign those values. More specifically, suppose variable $x_i$ is node inconsistent. In other words, a value $v_i \in d_i$ violates the unary constraint on $x_i$. Whenever BT tries to assign $v_i$ to $x_i$, the assignment will violate the unary constraint on $x_i$ and the search backtracks to the previous variable. Therefore, the value assignment of $v_i$ to $x_i$ keeps failing for the same reason.

Another cause of thrashing is that a value assigned to a variable $x_i$ in the ordering may conflict with all values of a variable $x_j$ which comes after $x_i$. For example, suppose the ordering of variable instantiation is $x_1, \cdots, x_i, \cdots, x_j, \cdots, x_n$. For a value $v \in d_i$, if a constraint forbids any value being assigned to $x_j$ whenever $v$ is assigned to $x_i$, then each time after BT assigns $v$ to $x_i$, it will backtrack later when trying to assign any value to $x_j$, thus leading to unnecessary assignments of values to $v_k, i < k < j$. In general, BT does not remember inconsistencies it found in previous branches of the search tree because BT backtracks to the most recent variable, rather than to a variable that is responsible for the inconsistent assignment.

### Backjumping, Backmarking, and the Hybrid of Backjumping and Backmarking

In order to prevent thrashing, three algorithms based on BT are developed by remembering the reasons for past backtrackings. Each algorithm avoids thrashing by either looking forward or looking back along the current partial assignment which can provide valuable information about possible reasons for thrashing.

The first method is Backjumping (BJ) (Prosser, 1993). When a partial assignment violates some constraints, BJ always jumps back to the *culprit*, which is the most recently instantiated predecessor found incompatible with any of the newly instantiated variable's values. So, BJ avoids the unnecessary repetition of instantiation failures which will be encountered in BT. For example, suppose the variable ordering is $x_1, \cdots, x_i, \cdots, x_j, \cdots, x_n$. For a value $v \in d_i$, if a constraint forbids any value being assigned to $x_j$ whenever $v$ is assigned to $x_i$, then each time BJ tries to assign any value to $x_j$ after $v$ is assigned to $x_i$, it will backtrack to $x_i$, thus avoiding unnecessary repetitive assignment of values to $x_k, i < k < j$.

The second method is Backmarking (BM) (Prosser, 1995). After a new instantiation is made,

BM avoids checking the constraints which have already been checked in an earlier instantiation. There are two types of unnecessary constraint checks. One is known to fail, and the other is known to succeed. In the case of unnecessary constraint checks that are known to fail, suppose we are instantiating a new value $v$ to the current variable $x_i$. Then, we check the value assignment $x_i = v$ against all previous value assignments. Namely, we check $x_i = v$ against $x_1 = a \in d_1$ first. If the check succeeds, we continue to check $x_2 = b \in d_2$ and so on, until the check fails. Assume that the constraint check between $x_h = g \in d_h$ and $x_i = v$ fails ($h < i$), we can deduce from this point that: If the next time we are instantiating $x_i = v$ again and $x_h$ has not been re-instantiated to another value, then the constraint check between them will fail. So, there is no need to perform this constraint check and we can backtrack. In the case of unnecessary constraint checks that are known to succeed, suppose since last time we visited $x_i$ we have backtracked to $x_j$ (j < i), and we have re-instantiated all the variables from $x_j$ to $x_{i-1}$ again. Then we know that the constraint checks between $x_k$ and $x_i$ are going to succeed, for all $k < j$. So, we can avoid those constraint checks and only check the constraints between $x_l$ and $x_i$, for all $j \leq l < i$.

Experimental results showed that BM and BJ are more efficient than GT and BT on solving CSPs. Nadel (Nadel, 1989) suggested combining BM and BJ and Prosser (Prosser, 1993) implemented the hybrid BMJ algorithm. However, Prosser found that BMJ does not inherit all the power from BM and BJ. BMJ may even perform worse than BM "*because the advantage of backmarking may be lost when jumping back*" (Prosser, 1993).

### 2.2.2 Consistency Propagation

Consistency Propagation (CP) is another popular method for solving CSPs, because it removes inconsistent values from domains, thus reducing the search space. A binary constraint problem can be modelled as a constraint graph $G = (V, E)$, where $V$ is a set of nodes, and $E$ is a set of edges. Each $v \in V$ represents a variable and each $e \in E$ represents a constraint between two variables. For constraint problems with $k$-ary constraints where $k > 2$, a hyper-graph can be used to represent it.

The consistency can be classified into three levels: node-consistency, arc-consistency, and k-

consistency ($k > 2$). Node-consistency propagation prunes inconsistent values by checking the unary constraints. Arc-consistency propagation removes inconsistent values by checking binary constraints. K-consistency propagation removes values by checking $m$-ary constraints ($2 \leq m \leq k$). Usually, node-consistency and arc-consistency propagations cannot guarantee that a solution can be found without search in general. If a $n$-node constraint graph is $n$-consistent, then a solution can be found without any backtracking. Unfortunately, the time complexity to enforce $n$-consistency in a $n$-node constraint graph is exponential in $n$.

### Node and Arc Consistency

The following definitions are based on concepts introduced by Mackworth (1977). They are the foundations of all the techniques we will discuss in this thesis.

**Definition 2.5.** *Node Consistency . A value $v_i \in d_i$ is node consistent if it is permitted by the unary constraint $c_i$ defined over $x_i$. $x_i$ is node consistent if all of its values are node consistent. A CSP is node consistent if all of its variables are node consistent.*

**Definition 2.6.** *Arc Consistency . Suppose a binary constraint $c \in C$ is represented by an arc $(x_i, x_j)$ in the constraint graph. The constraint is arc consistent relative to $x_j$ if for every value $v \in d_i$, there is some value $w \in d_j$ such that the value combination $(v, w)$ is permitted by the constraint $c$. Value $w$ is called a support for the value $v$.*

To make a variable node consistent, all values violating the unary constraint of the variable have to be deleted. To make a binary constraint $c_{ij}$ arc consistent relative to $x_j$, all values from $d_i$ which do not have any support in $d_j$ have to be deleted.

Waltz (Waltz, 1975) initiated the research on constraint propagation algorithms by introducing an algorithm for three-dimensional cubic drawing interpretations. His empirical results demonstrated that for some polyhedral problems, basic arc consistency algorithms are able to solve them. The basic operation is $REVISE$ (Mackworth, 1977) which prunes values from $d_i$ to make the constraint $c_{ij}$ arc consistent relative to $x_j$. To make a CSP arc consistent, it is not sufficient to perform $REVISE$ on each constraint just once, because the pruning of values may make some

arc consistent constraints inconsistent, if the pruned values are the only supporting values for some other domain's remaining values. AC1 (Mackworth, 1977) is a simple intuitive algorithm for consistency propagation. It repeats $REVISE$ on each arc until no domains are changed or some domain becomes empty. AC1 is not efficient because it checks all the arcs again even if only one arc is changed. AC3 (Mackworth, 1977) improves AC1 by maintaining a queue of constraints waiting to be processed, thus checking only the constraints for variables whose supports may have changed. Other versions of consistency propagation include AC2 (Mackworth, 1977) which is a special case of AC3, AC4 (Mohr and Henderson, 1986) which builds additional data structures to simplify the propagation process, and so on.

Arc consistency (AC) can be interpreted as an operation transforming a problem into an equivalent one, possibly with some domain values removed, in order to reduce the search space. When using AC for preprocessing before the search starts, if some domains become empty, then there is no solution. However, AC does not completely eliminate the need for search. If there is no empty domain and at least one domain has more than one value after AC is done, we still have to search the remaining constraint graph to find a solution.

A powerful hybrid way to solve CSPs is to combine consistency propagation (CP) and search, in which AC propagation is performed at each new value instantiation during the search. When a value is assigned to a variable, the domain of this variable is reduced to contain only this value. Then AC propagation removes values without support. For example, suppose the variable ordering is $\{x_1, \cdots, x_i, \cdots, x_n\}$, and variables $x_1, \cdots, x_i$ have already been assigned values. When a value $v \in d_{i+1}$ is assigned to $x_{i+1}$, AC propagates over the constraints in the constraint graph to delete values without support. If AC propagation leads to empty domains, the search procedure will try to assign another value to $x_{i+1}$ or backtrack to $x_i$ if there is no more value available in $x_{i+1}$.

Although AC propagation can reduce the search space substantially, it requires more constraint checks than exhaustive search during each new value instantiation. There is a trade-off between the AC propagation and the search.

### 2.2.3 Heuristics

A heuristic is a procedure that guesses the best option based on information available during problem-solving: Instead of pursuing a definite correct answer, a heuristic tries to find a good enough approximation to the best answer. Heuristics are designed to increase computational performance, probably at the cost of accuracy or precision. Due to the efficiency of heuristics, they are used in many real world applications. For example, some medical expert systems use heuristics to deduce approximate diagnoses of a disease.

In tree search, a heuristic is used to choose variables and instantiate values. Instead of choosing the globally best variable or value at each instantiation point, which is expensive, a good enough heuristic is a relatively inexpensive procedure that tries to guide the search to the answer as fast as possible. For solving CSPs, there are two typical uses of heuristics, namely variable ordering and value ordering.

A variable-ordering heuristic re-orders the variables for instantiation in tree search, instead of using an arbitrary ordering (see Page 31). For example, the algorithm *search rearrangement* always chooses a variable with the least number of values remaining as the next instantiation, hoping to force backtracks to happen as early as possible (Bitner and Reingold, 1975). Compared with a lexicographical ordering, this method may traverse fewer nodes because a backtrack is likely to be triggered earlier in the search. However, this method can be ineffective in some cases, especially at the beginning of the search, where all the variables have the same remaining domain size. In order to avoid this pitfall, a static ordering is introduced which chooses a variable with the minimum remaining domain values and breaks ties by choosing the variable which has the largest number of constraints connected to future variables in the search tree (Brélaz, 1979). If multiple variables have the same remaining domain size and constrain the same number of future variables, additional tie breakers will break the ties by the domain size of the smallest neighbour and the number of triangles in which the first chosen variable is involved (Smith, 1999). Another famous variable ordering is *cycle-cutset decomposition* (Dechter, 1990). Since a tree-structured CSP can be solved in linear time without backtracking once it is node and arc consistent (Kumar, 1992), the

cycle-cutset heuristic tries to find a small set of variables whose removal makes the constraint graph tree-structured, and orders those variables earlier in the search tree than other variables. A variable can be removed from the constraint graph by assigning a value to it. However, finding the smallest cutset is NP-hard, so the cycle cutset decomposition is incorporated into tree search by instantiating variables unchanged after enforcing consistency, and triggering a specialized tree-solving algorithm on the remaining graph once a tree structure is encountered (Dechter, 1990). This method is not guaranteed to find a minimum cycle cutset, but it is a fast heuristic that is able to find a good approximation. If the constraint graph is complete, then the cycle cutset decomposition reverts to naive backtracking.

A value-ordering heuristic re-orders the values to be assigned to the next variable. For example, the value-ordering heuristic introduced by Dechter and Pearl (1988) approximates the number of possible solutions in the subtree associated with each value of the current variable and chooses the value with the highest number to instantiate the next variable. Vernooy and Havens (1999) introduced another dynamic value-ordering heuristic which decomposes a CSP into a disjoint set of spanning trees and uses Bayesian networks to approximate solution probabilities for different values based on the current search state. xAC (Section 3.3) can be treated as a dynamic value ordering heuristic for COPs.

### 2.2.4 Local Search

Local search algorithms start with a random initial full assignment. These algorithms then explore other full assignments most of whose values are the same with the initial assignment except for a few. These full assignments are called a local neighbourhood. From this local neighbourhood, a best assignment which maximizes the number of satisfied constraints is selected. Then these local search algorithms restart the neighbourhood exploration procedure to find a better assignment until a local maximum (i.e., the best assignment does not change after restart) is reached. Since a local maximum is not guaranteed to be the global maximum, the local search algorithms either restart from another random full assignment or considers other locally sub-optimal assignments as well as

the locally optimal one (Selman *et al.*, 1992).

Compared with tree search and consistency propagation, local search is incomplete because it cannot guarantee to find a solution or prove there is no solution. Empirical analysis has shown that the performance of local search is strongly influenced by the number of solutions and problem hardness. For local search algorithms, the hardest CSPs usually have few solutions and occur during the solubility phase transition (Clark *et al.*, 1996). Generally speaking, a CSP is considered easy to solve if there are too many solutions or the problem is highly over-constrained. On the other hand, it is considered hard to solve when the number solutions is close to one.

### 2.2.5 Summary of CSP Algorithms

Section 2.2 reviewed tree search and AC propagation for solving CSPs, examined the hardness of CSPs and empirical evaluation of different algorithms, and discussed several variable and value ordering heuristics.

Tree search algorithms are complete, but inefficient. A simple backtracking algorithm does not learn from the failure of different nodes, thus leading to repetitive instantiations of the same set of variables. Therefore, AC propagation is performed at each node of the search tree to reduce the search space, thus improving the performance. Other techniques to improve the efficiency of tree search include learning the reason for failure and choosing the right variable or value ordering. Similar to CSPs, many algorithms for solving COPs adopt and modify successful CSP algorithms. The following sections introduce the work in COP and focus on the soft arc consistency algorithms (see Chapter 3).

## 2.3 Constraint Optimization Problems

Although many real world problems are perfect CSP instances, some of the problems require us to find an assignment of values to variables that has an optimal value, either maximum or minimum. In order to solve these problems, several extensions of the CSP framework were proposed by taking into account priorities (Schiex, 1992; Borning *et al.*, 1989), costs (Shapiro and Haralick, 1981),

uncertainties (Rosenfeld *et al.*, 1976), preferences (Rosenfeld *et al.*, 1976), etc. These frameworks address problems which are called Constraint Optimization Problems (COPs).

### 2.3.1 Hard and Soft Constraints

A classical CSP either allows a tuple in a constraint or forbids it, with no choice of expressing degrees of satisfaction. In a real world problem, however, we need to represent various levels of satisfaction, such as degrees of preference, or costs. Constraints can be categorized into three groups (Schiex *et al.*, 1995):

- "Hard" constraints: properties which have to be satisfied in all cases, e.g., physical properties;
- Preferences: properties which should be satisfied;
- Uncertainties: properties that are relevant in some situations which cannot be predicted with certainty; such properties may be ignored or represented as constraints.

The "soft" constraints include the second and third groups. In order to utilize "soft" constraints, a new methodology is introduced to transfer the violation of these "soft" constraints into a *specific criterion* that should be minimized (Schiex *et al.*, 1995). In the following sections, we will review the concepts of VCSPs, SCSPs, and WCSPs, which are the mathematical models for "soft" constraints.

### 2.3.2 Valued Constraint Satisfaction Problems

Due to various formulations of "soft" constraints, each of which uses its own operators and interpretation of the violation of constraints, an ordered commutative monoid is introduced to encompass most "soft" CSP extensions (Schiex *et al.*, 1995). A monoid is an algebraic structure with a single associative binary operation and an identity element. For example, the natural numbers form a commutative monoid with addition as the commutative binary operation and zero as its identity element (the integers also form a monoid with multiplication as the binary operation, and one as its identity element).

As a recap, a classical CSP is defined as a tuple $P = (X, D, C)$, where $X$ is a set of variables, $D$

is a set of finite domains each of which corresponds to a variable, and $C$ is a set of constraints each of which forbids certain value combinations of the variables involved in the constraint. A constraint can also be viewed as a pair $c_i = \langle s_i, r_i \rangle$, where $s_i$ is called the scope of $c_i$, and $r_i$ is a relation defined on $s_i$. The scope $s_i$ includes the variables affected by $c_i$. Suppose $s_i = \{x_{i_1}, \cdots, x_{i_r}\}$, then $r_i \subseteq d_{i_1} \times \cdots \times d_{i_r}$. A solution is an assignment of values to all the variables satisfying all constraints.

To express "soft" constraints, instead of allowing absolute satisfaction or violation, valuation is used to generalize the degree of preference or cost. Tuples in a constraint are associated with elements taken from a valuation structure:

**Definition 2.7.** *(Schiex* et al.*, 1995) A valuation structure $S = \langle E, \circledast, > \rangle$ consists of:*

- *$E$ is a set, whose elements are called valuations, which are totally ordered by $>$, with a maximum element $\top$ and a minimum element $\bot$;*
- *$\circledast$ is a commutative, associative closed binary operation on $E$:*
  - *Identity: $\forall a \in E, a \circledast \bot = a$;*
  - *Monotonicity: $\forall a, b, c \in E, (a \geq b) \Rightarrow ((a \circledast c) \geq (b \circledast c))$;*
  - *Absorbing element: $\forall a \in E, (a \circledast \top) = \top$.*

**Definition 2.8.** *(Schiex* et al.*, 1995) Strict monotonicity. $\forall a, b, c \in E$, if $(a > c), (b \neq \top)$, then $(a \circledast b) > (c \circledast b)$.*

Strict monotonicity is very useful since the quality of a potential solution is determined by combining valuations of the assignments associated with that solution. However, strict monotonicity is too restrictive for certain classes of COPs.

**Definition 2.9.** *(Schiex* et al.*, 1995) A binary operation $\circledast$ is idempotent if $\forall a \in E, a \circledast a = a$.*

Idempotency is incompatible with strict monotonicity once $E$ has more than two elements. To see this, we have $\forall a \in E, a \circledast \bot = a$, according to identity. Therefore, $\forall a \in E$ such that $\bot < a < \top$,

strict monotonicity implies that $(\bot \circledast a) < (a \circledast a)$, which implies that $a < (a \circledast a)$. Clearly, $a < (a \circledast a)$ means that the binary operation $\circledast$ is not idempotent.

Based on Definition 2.7, a valued CSP is defined as follows.

**Definition 2.10.** *(Schiex* et al.*, 1995) A Valued CSP (VCSP) is defined by a classical CSP $(X, D, C)$, a valuation structure $S = \langle E, \circledast, > \rangle$, and a function $\phi$ from $C$ to $E$. A VCSP is denoted $(X, D, C, S, \phi)$, and $\phi(c)$ is called the valuation of $c$.*

The valuation of an assignment is a combination of valuations of involved constraints, each of whose scopes is a subset of the variables in $X$. The combination is calculated using $\circledast$.

**Definition 2.11.** *(Schiex* et al.*, 1995) Given a VCSP $\mathcal{P} = (X, D, C, S, \phi)$ and a partial assignment $A$ of the variables $Y \subset X$, the valuation or cost of $A$ with respect to $\mathcal{P}$ is defined by:*

$$\mathcal{V}_{\mathcal{P}}(A) = \underset{c \in C, \text{A violates c}}{\circledast} [\phi(c)]$$

In Definition 2.11, the valuation of a partial assignment is the combination of the valuations of all the constraints which violate the partial assignment. This valuation of partial assignment can be interpreted as the cost or preference of the partial assignment.

The valuations in $E$ represent degrees of inconsistency or consistency. Usually, the valuation of an assignment is interpreted as the level of inconsistency; the higher the valuation, the worse the assignment. Therefore, a COP can be solved by finding an assignment $A$ with a minimum valuation, such that $\mathcal{V}_{\mathcal{P}}(A) \leq \mathcal{V}_{\mathcal{P}}(B)$, for all possible assignments $B$. Notice that $\bot$ represents complete consistency and $\top$ complete inconsistency, thus a lower valuation is preferred to a higher one.

### 2.3.3 Semiring-Based Constraint Satisfaction Problems

A Semiring-Based CSP (SCSP) is another framework subsuming all "soft" extensions of classical CSPs in order to model COPs (Bistarelli *et al.*, 1999). A SCSP is based on a semiring structure (see Definition 2.12), which consists of a set of valuations plus two operators. More specifically, these valuations describe degrees of consistency which can be interpreted as preferences or costs.

There are two operators which define how to combine constraints in order to generate a combined valuation or preference level.

The original definition of SCSPs (Bistarelli *et al.*, 1999) specified two extreme elements: **0** represents the worst or the least preferred choice and **1** represents the best or the most preferred choice. Since the valuation structure in VCSPs is a specific semiring, we use $\top$, the worst element in VCSP, to represent **0**; $\bot$, the best element in VCSP, to represent **1**. We modified the original definitions (Bistarelli *et al.*, 1999). Our definitions are as follows.

**Definition 2.12.** *A semiring is a tuple* $(A, +, \times, \top, \bot)$ *such that*

- $A$ *is a set;*
- *there are two extreme elements in* $A$*:* $\top$ *and* $\bot$*, where* $\top$ *represents the worst or the least preferred choice and* $\bot$ *represents the best or the most preferred choice;*
- $+$ *is the additive operation, which is a closed (i.e.,* $a, b \in A$ *implies* $a + b \in A$*), commutative (i.e.,* $a + b = b + a$*), and associative (i.e.,* $a + (b + c) = (a + b) + c$*) operation such that* $a + \top = a = \top + a$*, where* $\top$ *is the unit element of* $+$*;*
- $\times$ *is the multiplicative operation, which is a closed and associative operation such that* $\bot$ *is its unit element and* $a \times \top = \top = \top \times a$*.* $\top$ *is the absorbing element;*
- $\times$ *distributes over* $+$*, i.e.,* $a \times (b + c) = (a \times b) + (a \times c)$*.*

**Definition 2.13.** *A c-semiring is a semiring such that* $+$ *is idempotent (i.e.,* $a \in A$ *implies* $a + a = a$*),* $\times$ *is commutative, and* $\bot$ *is the absorbing element of* $+$*.*

Since $+$ and $\times$ are generic operators, we can assign any semantics to them. However, there are some common requirements for these operators, no matter what the actual semantics are. For example, $+$ is used to define a partial ordering $\leq_S$: $a \leq_S b$ if and only if $a + b = b$. $a \leq_S b$ means b is at least as good as a. Notice that the idempotency of $+$ is necessary to define $\leq_S$, because $a \leq_S a$ if and only if $a + a = a$. Using the partial ordering $\leq_S$, we can define the best solution. Notice that both $+$ and $\times$ are monotonic on $\leq_S$, i.e., $a \leq_S b, c \in A$ and $c$ is not an absorbing element implies $a + c \leq_S b + c$ and $a \times c \leq_S b \times c$.

In a c-semiring, $\perp$ is the absorbing element of the additive operation, i.e., $a + \perp = \perp$, which implies $\forall a, a \leq_S \perp$. Therefore $\perp$ is the maximum or the best element in $A$. Similarly, $\top$ is the minimum or worst element of the ordering $\leq_S$ because $\forall a$, $\top + a = a$ implies that $\top \leq_S a$. Therefore, $\forall a \in A$, we have $\top \leq_S a \leq_S \perp$. According to the monotonic property of $\times$ over $A$, $b \leq_S \perp$ implies $b \times a \leq_S \perp \times a$ which implies $a \times b \leq_S a$. So, the $\times$ operation is *extensive*, because $a \times b \leq_S a$. Intuitively, combining two valuations always results in one that is no better than either of them.

In the following sections, $\times$ may be closed on a certain finite subset of the c-semiring.

**Definition 2.14.** *Given any c-semiring $S = (A, +, \times, \top, \perp)$, and a finite set $I \in A$. $\times$ is closed on $I$; if $\forall a, b \in I$, $a \times b \in I$.*

In order to incorporate a semiring into the framework of CSPs, constraint systems are introduced (Bistarelli *et al.*, 1999). A constraint system includes a c-semiring, a set of variables, and a set of domains corresponding to these variables. A constraint associates an element in the c-semiring to a tuple in the constraint relation. Similar to a valuation in a valuation structure in VCSPs, this element in a c-semiring can be interpreted as a cost or a preference. A constraint problem consists of a constraint system and a set of constraints, plus a selected set of variables. Notice that this set of variables may not be the set of all variables, because we may want to assign values to a subset of all variables.

**Definition 2.15.** *(Bistarelli* et al.*, 1999) A constraint system is a tuple $CS = \langle S, D, V \rangle$, where $S$ is a c-semiring, $D$ is a finite set, and $V$ is an ordered set of variables. Given a constraint system $CS = \langle S, D, V \rangle$, where $S = (A, +, \times, \top, \perp)$, a constraint over $CS$ is a pair $\langle \text{def}, \text{con} \rangle$, where* $\text{con} \subset V$ *and* $\text{def} : d^k \rightarrow A$ *where $k$ is the size of con or the number of variables in it. con is called the type of the constraint, and* def *is called the value of the constraint. Moreover, a constraint problem $P$ over $CS$ is a pair $P = \langle C, \text{con} \rangle$, where $C$ is a set of constraints over $CS$ and* $\text{con} \subset V$*. If $\times$ is not idempotent, then $C$ becomes a multi-set.*

Each constraint associates a tuple of domain values to a valuation in the c-semiring. Figure 2.2

is a graphical representation of a constraint. In a graphical representation, a variable is a node and a constraint is an arc. Domains and constraints are labels of the corresponding graphical objects.

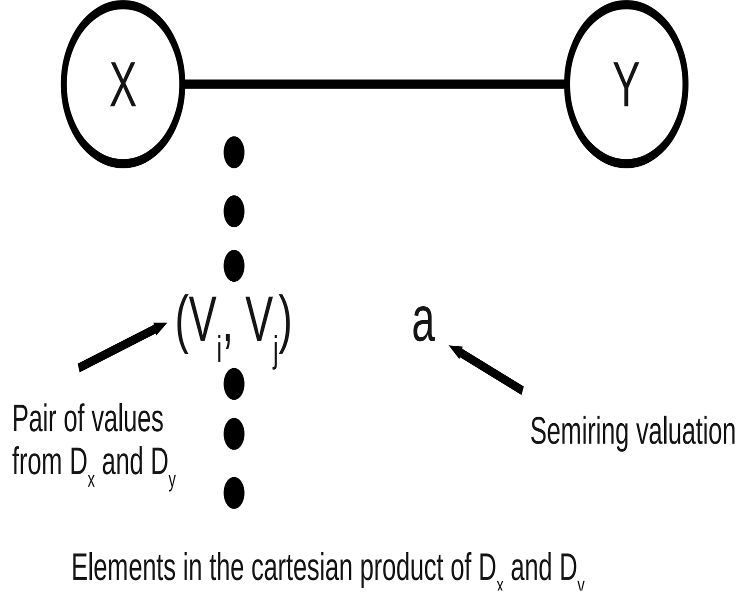



**Figure 2.2:** Structure of a Constraint

The following discussion uses a constraint system $CS = \langle S, D, V \rangle$, where $S = (A, +, \times, \top, \bot)$. A special set of variables includes the variables of every constraint: $V(P) = \cup_{\langle \text{def}, \text{con}' \rangle \in C} \text{con}^{'}$.

Since the elements in $A$ are associated with tuples of constraints, an SCSP's constraints can be manipulated by $\times$ and $+$. More specifically, two operations, *combination* $\otimes$ and *projection* $\Downarrow$, are defined using $\times$ and $+$.

**Definition 2.16.** *For any tuple $t = (t_1, t_2, \cdots, t_n)$ in the cartesian product of domain values in a variable set $I$, given another variable set $I^{'}$, the projection of $t$ over the variables in the set $I^{'}$ is defined as $t \downarrow^{I}_{I'} = (t^{'}_1, t^{'}_2, \cdots, t^{'}_m)$, such that any $t^{'}_i$ is a domain value of $d^{'}_i$, where $x^{'}_i \in I^{'} \cap I$.*

**Definition 2.17.** *(Bistarelli* et al.*, 1999) Consider two constraints $c_1 = \langle \text{def}_1, \text{con}_1 \rangle$ and $c_2 = \langle \text{def}_2, \text{con}_2 \rangle$ over $CS$. Their combination, $c_1 \otimes c_2$, is the constraint $c = \langle \text{def}, \text{con} \rangle$ with* $\text{con} = \text{con}_1 \cup \text{con}_2$

*and* $\mathrm{def}(\mathrm{t}) = \mathrm{def}(\mathrm{t} \downarrow^{\mathrm{con}}_{\mathrm{con}_1}) \times \mathrm{def}(\mathrm{t} \downarrow^{\mathrm{con}}_{\mathrm{con}_2})$*, where, for any tuple* $t$ *in a set* $I$, $t \downarrow^{I}_{I'}$ *denotes the projection of* $t$ *over the variables in the set* $I^{'}$*. Moreover, given a constraint* $c = \langle \mathrm{def}, \mathrm{con} \rangle$ *over* $CS$*, and a subset* $w$ *of* con*, its projection over* $w$*,* $c \Downarrow_w$*, is the constraint* $\langle \mathrm{def}^{'}, \mathrm{con}^{'} \rangle$ *over* $CS$ *with* $\mathrm{con}^{'} = \mathrm{w}$ *and* $\mathrm{def}^{'}(\mathrm{t}^{'}) = \sum_{\{t | t \downarrow^{con}_{w} = t'\}} \mathrm{def}(\mathrm{t})$.

Figure 2.3 shows an example of combination and projection. A solution of an SCSP can now be defined using these two operations.

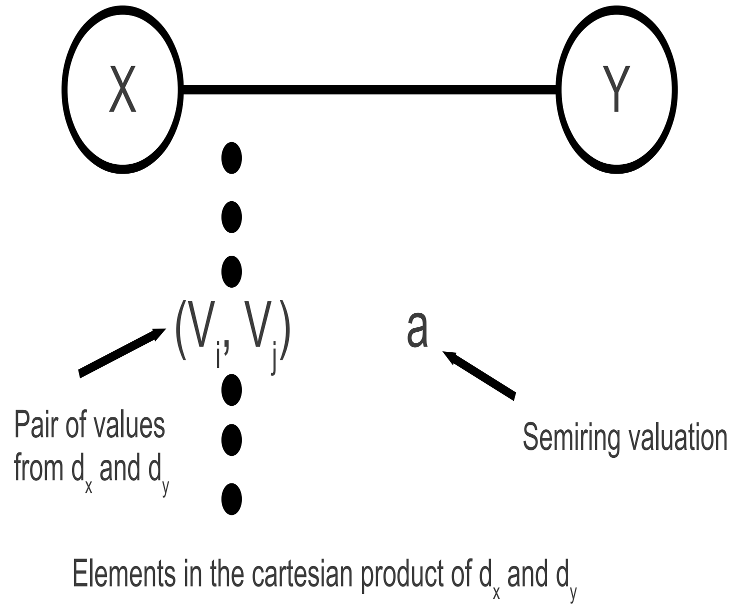



**Figure 2.3:** An Example of Combination and Projection

**Definition 2.18.** *(Bistarelli* et al.*, 1999) Given a constraint problem* $P = \langle C, con \rangle$ *over a constraint system* $CS$*, the solution of* $P$ *is a constraint defined as* $Sol(P) = (\bigotimes C) \Downarrow_{con}$*, where* $\bigotimes C = c_1 \otimes c_2 \otimes \cdots \otimes c_n, C = \{c_1, c_2, \cdots, c_n\}$*. The entity including the constraint problem* $P$ *and the constraint system* $CS$ *is called an SCSP.*

In other words, the solution of an SCSP is an induced global constraint which combines all constraints in the problem and associates valuations in $A$ to each tuple of values of $D$. Notice that

Definition 2.18 specifies the semantic of the solution of a problem, not how to solve it.

### 2.3.4 Comparison of SCSPs and VCSPs

Given a variable ordering, it is possible to transform any SCSP into an equivalent VCSP, and vice-versa (Bistarelli *et al.*, 1999). These transformations are explained in the following sections.

#### From SCSPs to VCSPs

An SCSP $\langle C, con \rangle$ where *con* involves all variables is considered in the following sections. Recalling the definition of SCSP, it is a set of constraints $C$ over a constraint system $\langle S, D, V \rangle$, where $S = (A, +, \times, \top, \bot)$ is a c-semiring, $D$ is a set of domains, each of which corresponds to a variable in $V$. For this section, assumptions are made that $+$ induces a total ordering $\leq_S$, which indicates that $+$ corresponds to an operator that always chooses a valuation closer to $\bot$ among any two valuations because $\bot$ means total consistency (Bistarelli *et al.*, 1999).

Given an SCSP, an equivalent VCSP assigns the same valuations to tuples, and has the same solutions (Bistarelli *et al.*, 1999). For example, Figure 2.4 shows an SCSP which contains a constraint $c = \langle con, def \rangle$, where $con = \{x, y\}$, and $def(\langle a, a \rangle) = 1, def(\langle a, b \rangle) = 3, def(\langle b, a \rangle) = 1, def(\langle b, b \rangle) = 2$. Then, the equivalent VCSP will contain three constraints, all of which constrain $x$ and $y$:

- $c_1$, with $\phi(c_1) = 1$ and allowed tuples $\langle a, b \rangle$ and $\langle b, b \rangle$.

- $c_2$, with $\phi(c_2) = 3$ and allowed tuples $\langle a, a \rangle$, $\langle b, a \rangle$, and $\langle b, b \rangle$.

- $c_3$, with $\phi(c_3) = 2$ and allowed tuples $\langle a, a \rangle$, $\langle a, b \rangle$, and $\langle b, a \rangle$.

No matter how an SCSP is constructed, there is always an equivalent VCSP. In our experimentations (see Chapter 4, Chapter 5, and Chapter 6), xAC solves SCSPs, while W-AC*2001, EDAC, and VAC solve VCSPs.

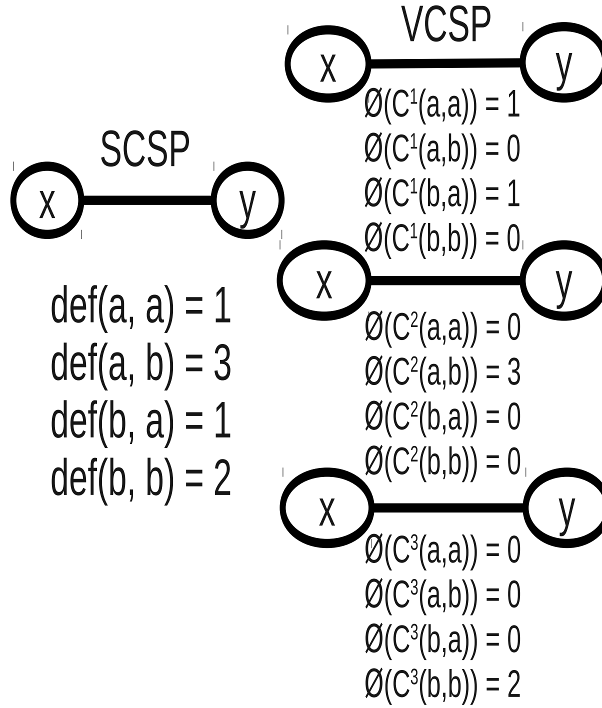



**Figure 2.4:** From SCSP to VCSP

**From VCSPs to SCSPs**

In this section, we translate a VCSP to an equivalent SCSP (Bistarelli *et al.*, 1999). For example, Figure 2.5 shows a VCSP with $\perp = 0$, $\top = M$, where $M$ is a finite integer, and $\geq_S$ is given the semantics of $\leq$ over integers. This VCSP contains a binary constraint $c$ between variables $x$ and $y$. The constraint $c$ allows tuples $\langle a, a\rangle$ and $\langle b, b\rangle$, and such that $\phi(c) = 1$. Then the corresponding SCSP will contain the constraint $c^{'} = \langle con, def\rangle$, where $con = \{x, y\}$, $def(\langle a, a\rangle) = def(\langle b, b\rangle) = \perp = 0$, and $def(\langle a, b\rangle) = def(\langle b, a\rangle) = 1$. Notice we assume a cost of 0 equals $\perp$.

If there are multiple VCSP constraints defined over the same subset of variables, each of the constraints can be converted into an SCSP constraint and all of the SCSP constraints can be combined using the multiplicative operator (see Definition 2.12). For example, each of the VCSP constraints on the right hand side in Figure 2.4 can be converted into an SCSP constraint and the resulting SCSP constraints can be combined (i.e., all the costs of each tuple are sumed) to form

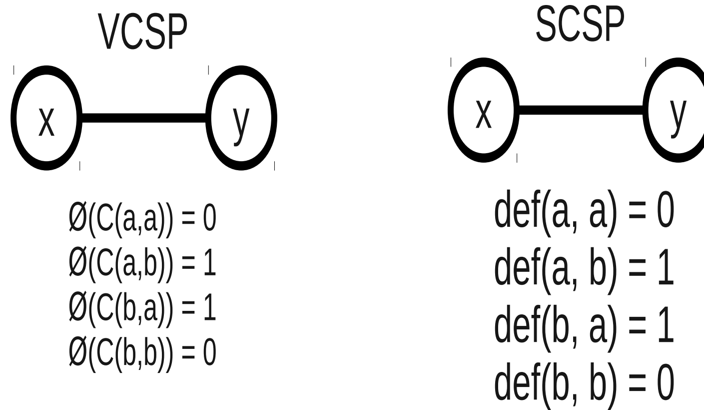



**Figure 2.5:** From VCSP to SCSP

the SCSP constraint on the left hand side.

### 2.3.5 Weighted CSPs

Both VCSP and SCSP can be used to model Weighted CSP (WCSP), which is another one of the main frameworks for COPs (Bistarelli *et al.*, 1999).

#### VCSP Model for WCSP

A Weighted CSP (WCSP) assigns weights or degrees of preferences to the tuples in constraints. These weights represent the costs of violating the constraints. Solving WCSPs means minimizing the weighted sum of the elementary weights associated with violated constraints among all potential solutions. VCSP models WCSP by giving the operation $\circledast$ the semantic of arithmetic addition on integers. The set $E$ in the valuation structure $S$ contains natural numbers plus positive infinity, which is $\mathcal{N} \cup \{+\infty\}$. The ordering over $E$ is the usual binary operator $<$ over natural numbers.

Notice that $\circledast$ is strictly monotonic.

**SCSP Model for WCSP**

In SCSP, a WCSP can be modelled as an SCSP with a c-semiring $S^{-}_{WCSP} = \langle \mathcal{R}^{-}, max, +, -\infty, 0\rangle$, where ordering $\leq_S$ is given the semantics of $\leq$ over real numbers. Or, the c-semiring can be $S^{+}_{WCSP} = \langle \mathcal{R}^{+}, min, +, +\infty, 0\rangle$, where ordering $\leq_S$ equals $\geq$ over real numbers. Notice that the former c-semiring represents costs as negative numbers and the latter represents them as positive numbers.

If a WCSP problem has a best solution with cost $\alpha$, then the best solution of any subproblem has a cost that is at least $\alpha$. Therefore, we can use the cost of the best solution found so far as the upper bound in a branch and bound search. If the current partial solution's cost is greater than $\alpha$, we can prune the current branch. Notice that the same properties hold for the semirings over rational costs and integer costs: $\langle \mathcal{Q}^{-}, max, +, -\infty, 0\rangle$ and $\langle \mathcal{Z}^{-}, max, +, -\infty, 0\rangle$ (Bistarelli *et al.*, 1999).

## 2.4 Summary

In this chapter, we reviewed the current frameworks for constraint satisfaction problems and constraint optimization problems. We demonstrated how Valued CSPs and Semiring CSPs are derived from classical CSPs and how to transform each one into an equivalent other. This chapter builds the theoretical foundation for the following chapters.

# Chapter 3

# Soft AC Algorithms

In this chapter, we briefly review algorithms W-AC*2001, EDAC, VAC, and xAC, all of which can be used to solve general COPs. W-AC*2001, EDAC, and VAC solve WCSPs modelled by VCSPs, while xAC solves WCSPs modelled by SCSPs. Since we have shown that both VCSPs and SCSPs can be used to model WCSPs in Section 2.3.5, we can model the same WCSPs with VCSPs or SCSPs, which produce the same solutions. We note that VAC is state-of-the-art for solving COPs (Cooper *et al.*, 2008). We use AC* to refer to W-AC*2001 in this chapter in order to show the connection between AC*, FDAC*, and EDAC*.

## 3.1 Preliminary

### 3.1.1 Conventions

In the literature, it is common for authors to use an acronym in two ways: one as an algorithm, the other as a property realized after the algorithm is applied. In this thesis, AC* refers to the algorithm W-AC*2001 or the arc consistent property W-AC*2001, depending on context.

### 3.1.2 Arc Consistency Closure

**Definition 3.1.** *(Robert, 1999) A set of objects, $O$, is said to exhibit closure or to be closed under a given operation, $R$, provided that for every object, $x$, if $x$ is a member of $O$ and $x$ is $R$-related to any object, $y$, then $y$ is a member of $O$.*

For example, real numbers are closed under the subtraction operation because the result of performing subtraction among any pair of real numbers is a real number. But natural numbers

are not closed under the subtraction operation because the result of performing subtraction among some pairs of natural numbers may be negative, which is not a natural number.

By applying Definition 3.1 to classical arc consistency (Mackworth, 1977), we get the following definition of arc consistency closure.

**Definition 3.2.** *Given a set of objects $O$ which is a set of constraints, a given operation $R$ which is the $REVISE$ operation (Mackworth, 1977), and every object $x$ which is an arc consistent constraint in $O$, a CSP is an arc consistency closure if the result of applying $R$ (i.e., $REVISE$) to $x$ is a constraint $y$ that is arc consistent.*

In other words, an arc consistency closure is a CSP whose domain values and solutions will not change no matter how many times the $R$ operation (i.e., the $REVISE$ operation in the context of classical CSPs) is applied. An arc consistency closure is closed under the operation $REVISE$.

Although applying the $REVISE$ operation multiple times in a classical CSP can reach a unique arc consistency closure (see Section 2.2.2), applying soft arc consistency operations (described in this chapter) is not guaranteed to reach a unique soft arc consistency closure. Therefore, people continue to research algorithms to produce better closures that can improve the efficiency of problem solving (Schiex, 2000; Larrosa, 2002; Cooper and Schiex, 2004; Givry and Zytnicki, 2005; Cooper *et al.*, 2008).

## 3.2 W-AC*2001 and EDAC

### 3.2.1 Foundation

Based on the notation of VCSP, a single axiom is introduced to define an extended arc consistency (AC) which has all the properties of a classical arc consistency except for the uniqueness of the arc consistency closure (Cooper and Schiex, 2004). An arc consistency closure is a COP obtained by removing all arc inconsistent values from the problem's domains. If the VCSP binary operation $+$ is idempotent, then the new AC closure reduces to classical definitions and uniqueness is recovered.

The new axiom added to the VCSP framework is based on an intuition that we can transfer a

VCSP into an equivalent one by shifting costs of constraints. The idea can be formalized, starting with the definition of a difference of two valuations:

**Definition 3.3.** *(Cooper and Schiex, 2004) In a valuation structure $S = \langle E, \oplus, \succeq \rangle$, if $\alpha, \beta \in E$, $\alpha \preceq \beta$, and there exists a valuation $\gamma \in E$ such that $\alpha \oplus \gamma = \beta$, then $\gamma$ is known as a difference of $\beta$ and $\alpha$. $\alpha \preceq \beta$ is equivalent to $\beta \succeq \alpha$.*

**Definition 3.4.** *(Cooper and Schiex, 2004) The valuation structure $S$ is* fair *if for any pair of valuations $\alpha, \beta \in E$, with $\alpha \preceq \beta$, there exists a maximal difference of $\beta$ and $\alpha$. This maximal difference of $\beta$ and $\alpha$ is denoted by $\beta \ominus \alpha$.*

**Theorem 3.1.** *(Cooper and Schiex, 2004) Let $S = \langle E, \oplus, \succeq \rangle$ be a fair valuation structure. Then $\forall u, v, w \in E$, $w \preceq v$, we have $(v \ominus w) \preceq v$ and $(u \oplus w) \oplus (v \ominus w) = (u \oplus v)$.*

Most existing soft constraint frameworks are fair. For example, in order to count costs as integer values, we can define a strictly monotonic valuation structure $\langle \mathcal{N} \cup \{\infty\}, +, \geq \rangle$. $\beta \ominus \alpha = \beta - \alpha$ for finite valuations $\alpha, \beta \in \mathcal{N}, \alpha \leq \beta$ and $(\infty \ominus \alpha) = \infty$ for all $\alpha \in \mathcal{N} \cup \{\infty\}$. Although $\ominus$ may not exist in general cases of any strictly monotonic operator $\oplus$, it can always be constructed by deriving a larger valuation structure $E \times E$, where each valuation $(\beta, \alpha)$ represents the imaginary $\beta \ominus \alpha$ (Cooper and Schiex, 2004). This is similar to extending real numbers $\mathcal{R}$ to complex numbers $\mathcal{C}$ in order to represent square roots of negative numbers.

### Equivalence Preserving Transformations

**Definition 3.5.** *A value assignment is a tuple $(i, a)$ which denotes assigning value $a \in d_i$ to variable $x_i \in X$. (In Chapter 2, we used a different notation for this.)*

**Definition 3.6.** *(Cooper and Schiex, 2004) The subproblem of a VCSP $V = \langle X, D, C, S \rangle$ on $J \subset X$ is a VCSP $V(J) = \langle J, D_J, C_J, S \rangle$, where $D_J = \{d_j : j \in J\}$ and $C_J = \{c_P \in C : P \subset J\}$.*

**Definition 3.7.** *(Cooper and Schiex, 2004) For a VCSP $V$, an equivalence-preserving transformation of $V$ on $J \subset X$ is an operation which transforms the subproblem of $V$ on $J$ into an equivalent*

*VCSP, which has the same J and solutions. If* $C_J = \{c_P \in C : P \subset J\}$ *contains only one non unary constraint, such an operation is called an equivalence-preserving arc transformation.*

Procedures 1 and 2 show two operations *Project* and *Extend* that transform a VCSP into an equivalence. For a subproblem $V(J) = \langle J, D_J, C_J, S\rangle$, $l(J)$ denotes the set of all possible value assignments for $J$ (i.e., the Cartesian product of all $d_i \in D_J$). Procedure 1 projects the minimum cost in a given non-unary constraint $c_P$ down to a value $a \in d_i$, where $i \in P, a \in d_i$. The procedure adds the minimum cost to the unary constraint of $c_i$ and subtracts that cost from the non-unary constraint $c_P$ in order to preserve equivalence. Conversely, Procedure 2 extends the cost from value $a \in d_i$ to the non-unary constraint $c_P$ by subtracting the cost $c_i(a)$ from the unary constraint $c_i$ and adding that cost to the non-unary constraint $c_P$.

**Procedure 1** Project($c_P, i, a$)

**Input:** a variable $i$, a value $a \in d_i$, and a constraint $c_P$
  $\beta \leftarrow min_{t \in l(P-\{i\})}(c_P(t,a))$
  $c_i(a) \leftarrow c_i(a) \oplus \beta$
  **for** each $t \in l(P - \{i\})$ **do**
    $c_P(t,a) \leftarrow c_P(t,a) \ominus \beta$
  **end for**

**Procedure 2** Extend($i, a, c_P$)

**Input:** a variable $i$, a value $a \in d_i$, and a constraint $c_P$
  **for all** $t \in l(P - \{i\})$ **do**
    $c_P(t,a) \leftarrow c_P(t,a) \oplus c_i(a)$
  **end for**
  $c_i(a) \leftarrow c_i(a) \ominus c_i(a)$

### 3.2.2 W-AC*2001 and Directional Arc Consistency

The Weighted Arc Consistency 2001 (W-AC*2001) (Larrosa, 2002) and the Full Directional Arc Consistency (FDAC*) refine the original soft AC (Schiex, 2000) to provide better guidance for a branch-and-bound search. The W-AC*2001 algorithm and the FDAC* algorithm are based on the VCSP framework introduced in Section 2.3.2.

**Local Consistency in WCSP**

In this section we review the definitions of consistency as applied to variables and constraints (Larrosa, 2002; Larrosa and Schiex, 2003). We assume the variable set $X$ is lexicographically ordered according to a sequence given by the problem statement.

**Definition 3.8.** *Let $P = (X, D, C, S, \phi)$ be a binary WCSP, where $X$ is a set of variables, $D$ is a set of domains, $C$ is a set of constraints, $S = (\mathcal{N} \cup \{+\infty\}, +, >)$ is a valuation structure, and $\phi$ is a function mapping a constraint tuple to a valuation in $S$. The maximum cost is denoted by $\top$ and the minimum cost is denoted by $\bot$.*

- *Zero-arity Constraint: $c_\emptyset$ is the zero-arity constraint which is usually the lower bound for a branch-and-bound search. This constraint's value indicates the least amount of cost for the optimal solution at any given point in the search.*

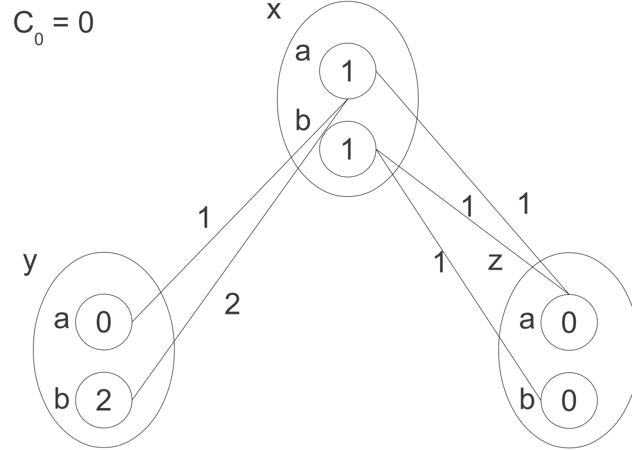
- *Node consistency ($NC^*$): The value assignment $(i, a)$ is node consistent ($NC^*$) if $c_\emptyset \oplus c_i(a) < \top$. Variable $i$ is $NC^*$ if: 1) all its values are $NC^*$, and 2) there exists a value $a \in d_i$ such that $c_i(a) = \bot$. Value $a$ is a support for the variable $i$. $P$ is $NC^*$ if every variable is $NC^*$.*

- *Arc consistency (AC): The value assignment $(i, a)$ is arc consistent (AC) with respect to constraint $c_{ij}$ if there is a value $b \in d_j$ such that $c_{ij}(a, b) = \bot$. Value $b$ is called a support of value $a$. Variable $i$ is AC if all its values are AC with respect to every binary constraint affecting $i$. $P$ is $AC^*$ if every variable is AC and $NC^*$.*

- *Directional arc consistency ($DAC^*$): Given a variable ordering such that $i < j$ if $i$ appears earlier in the ordering than $j$, the value assignment $(i, a)$ is directional arc consistent ($DAC^*$) with respect to constraint $c_{ij}$, where $j > i$, if there is a value $b \in d_j$ such that $c_{ij}(a, b) \oplus c_j(b) = \bot$. Value $b$ is called a full support of $a$. Variable $i$ is DAC if all its values are DAC with respect to every $c_{ij}, j > i$. $P$ is $DAC^*$ if every variable is DAC and $NC^*$.*

- *Full Directional Arc Consistency ($FDAC^*$). $P$ is fully directional arc consistent if it is $DAC^*$ and $AC^*$.*

**Figure 3.1:** A WCSP instance that is not $NC*$




For example, Figure 3.1 is a constraint graph which shows a WCSP with $\top = 4, \bot = 0$. In a constraint graph, small circles representing domain values are contained in larger circles which represent variables. Constraints are represented by edges showing non-zero cost tuples as labels. If there is no edge between two domain values, then there is zero cost associated with the tuple. Figure 3.1 is not $NC^*$ because variable $x$ does not have a value $a$ such that $c_x(a) = \bot$. We can make this WCSP $NC^*$ by projecting a unary cost of one from $x$ down to $c_\emptyset$. This unary projection subtracts one cost from all values of $x$ and adds one cost to $c_\emptyset$. The result is Figure 3.2.

Figure 3.2 is not $DAC^*$ with respect to variable ordering $xyz$ because the value $a$ of $x$ has no full supports in $y$, and the value $b$ of $x$ has no full supports in $z$. We can make it $DAC^*$ by projecting a binary cost of one from $c_{xy}$ down to $c_x$ and from $c_{xz}$ down to $c_x$. Then we project the a cost of one from $c_x$ down to $c_\emptyset$. The binary projections from $c_{xy}$ to $c_x$ and $c_{xz}$ to $c_x$ shift one cost from those binary constraints down to each value of $x$ and the unary projection from $c_x$ to $c_\emptyset$ moves that one cost down to the zero-arity constraint. The result is shown in Figure 3.3. It can be shown that Figure 3.3 is also $AC^*$. Therefore, Figure 3.3 is $FDAC^*$.

In Figure 3.3, a WCSP is made $AC^*$ by making it $DAC^*$. However, if we choose to make the previous WCSP (Figure 3.2) $AC^*$ by projecting binary costs from $c_{xy}$ to $c_x$ and $c_{xz}$ to $c_z$, then the result is shown in Figure 3.4. It is not $DAC^*$ because $(x, b)$ doest not have a full support in $z$.

**Figure 3.2:** A WCSP instance that is $NC*$

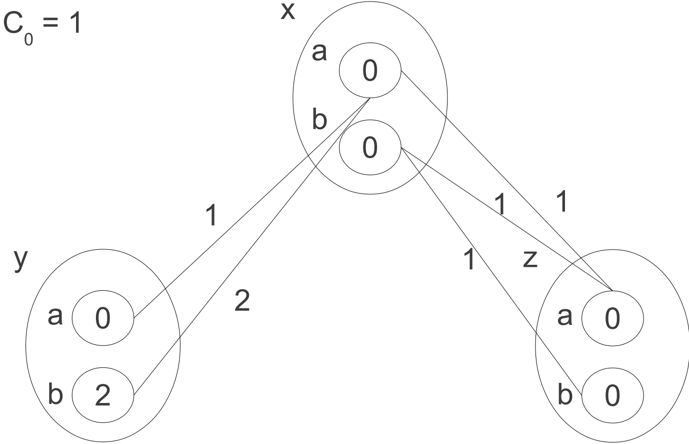



**Figure 3.3:** A WCSP instance that is $DAC*$

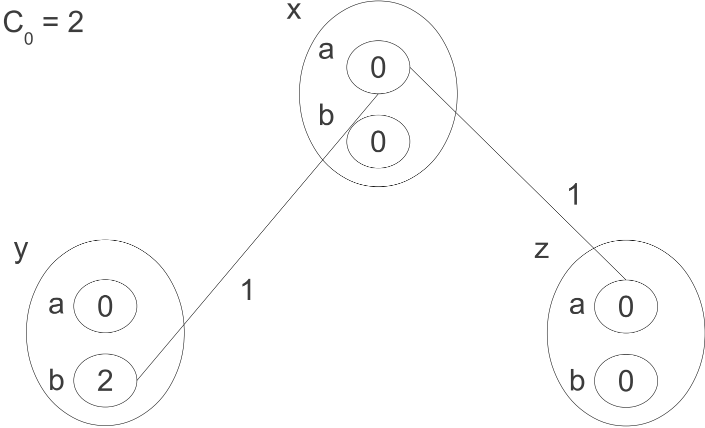
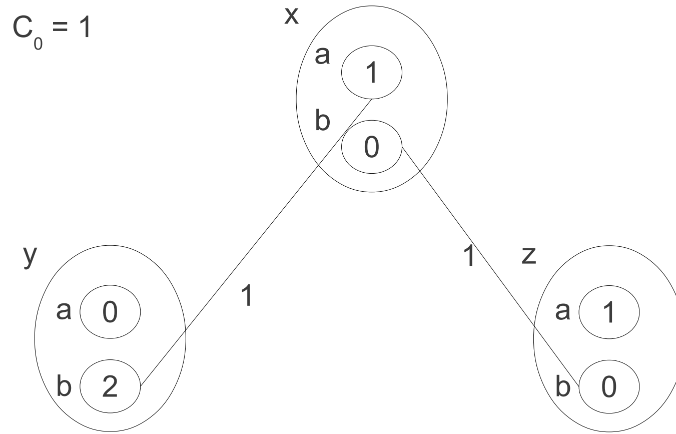


**Figure 3.4:** A WCSP instance that is $AC^*$ but not $DAC^*$




In WCSP instances, a support is not necessarily a full support. $DAC^*$ requires full supports on the variables later in the ordering while $AC^*$ requires a support on both variables. $FDAC^*$, which requires a support on one side and a full support on the other, is at least as strong as $AC^*$ or $DAC^*$. This property implies that the $c_\emptyset$ of $FDAC^*$ is equal to or stronger than that of $AC^*$ or $DAC^*$. In fact, $FDAC^*$ is stronger in many cases. One case is shown in Figure 3.3 and Figure 3.4 which demonstrate that $FDAC^*$ can produce a higher zero-arity constraint than $AC^*$.

### 3.2.3 Enforcing Arc Consistency

In the previous section, we showed an example of how to achieve $NC^*, AC^*, DAC^*, FDAC^*$ by shifting costs in the problem. In this section, we present algorithms to enforce these arc consistencies.

Let $P = (X, D, C, S, \phi)$ be a WCSP with a valuation structure $S = ([0, \cdots, k], +, >)$, where 0 represents the lowest cost or the most preferable valuation and $k$ represents the worst cost or the least preferable valuation. We use $+$ to combine costs or preferences and $>$ to compare them. Let

$a, b \in [0, \cdots, k]$, such that $a \geq b$. $a \ominus b$ is the difference between $b$ and $a$:

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

We present variants of *Project* (Procedure 1) and *Extend* (Procedure 2) which calculate and shift costs in the procedures. The variant *Projection* (Procedure 3) takes a cost of $\alpha$ given as an input, while Procedure 1 calculates a cost of $\beta$ within the procedure. The variant *Extension* (Procedure 4) takes a cost of $\alpha$ given as input, while Procedure 2 extends the full cost of $c_i(a)$ within the procedure. The *projection* of $\alpha$ cost units from $c_{ij} \in C$ to value $a \in d_i$ is a flow of $\alpha$ cost units from the binary constraint to the unary constraint $c_i(a)$. The *extension* of $\beta$ cost units from a value $a \in d_i$ to $c_{ij} \in C$ is a reverse flow of $\beta$ cost units from $c_i(a)$ to the binary constraint. It can be shown that *projection* and *extension* preserve the equivalence of the transformed WCSP (Larrosa and Schiex, 2003).

**Procedure 3** Projection$(i, a, j, \alpha)$

**Input:** variables $i$ and $j$, a value $a \in d_i$, and a cost $\alpha$
  $c_i(a) := c_i(a) \oplus \alpha$
  **for** each $b \in d_j$ **do**
    $c_{ij}(a, b) := c_{ij}(a, b) \ominus \alpha$
  **end for**

**Procedure 4** Extension$(i, a, j, \alpha)$

**Input:** variables $i$ and $j$, a value $a \in d_i$, and a cost $\alpha$
  **for** each $b \in d_j$ **do**
    $c_{ij}(a, b) := c_{ij}(a, b) \oplus \alpha$
  **end for**
  $c_i(a) := c_i(a) \ominus \alpha$

The $AC^*$ and $FDAC^*$ algorithms assume that no empty domain is produced and that the initial problem is $NC^*$. It is also assumed that no constraint includes more than two variables. The $AC^*$ algorithm requires two data structures $S(i, a, j)$ and $S(i)$. The data structure $S(i, a, j)$ stores the value support for $(i, a)$ with respect to constraint $c_{ij}$. The data structure $S(i)$ stores the value support for $i$. A value support is a value in a domain that provides support for a value in a domain.

A procedure can enforce supports for a value in a domain by shifting costs and finding some

value supports for that value. If a value without support becomes supported by some other values found by a procedure, this value's supports are enforced by that procedure.

The $AC^*$ algorithm requires three auxiliary functions and procedures. Procedure 5 projects unary costs from $c_i$ down to $c_\emptyset$. Function FindSupportAC*$(i, j)$ tries to find supports to each value of $i$ with respect to constraint $c_{ij}$ by projecting binary costs from $c_{ij}$ down to $c_i$. If a support can be found, this function returns true, otherwise it returns false. In other words, Function FindSupportAC*$(i, j)$ enforces supports to each value of $i$. The main algorithm $AC^*$ uses a queue $Q$ to store all the variables whose values have been pruned and these variables' neighbours will be examined for supports because the value deletion may lead to unsupported values. $Q$ is initialized to contain all the variables because a support should be found for every variable.

---

**Procedure 5** ProjectUnary$(i)$, where $i$ is a variable

**Input:** a variable $i$
  $S(i) := \arg\min_{a \in d_i}\{c_i(a)\}$
  $\alpha := c_i(S(i))$
  $c_\emptyset := c_\emptyset \oplus \alpha$
  **for** each $a \in d_i$ **do**
    $c_i(a) := c_i(a) \ominus \alpha$
  **end for**

---

**Function 6** FindSupportAC*$(i, j)$

**Input:** variables $i$ and $j$
**Output:** true if there is a support for $i$ from $j$, false otherwise
  $flag :=$ FALSE
  **for** each $a \in d_i$ s.t. $S(i, a, j) \notin d_j$ **do**
    $S(i, a, j) := argmin_{b \in d_j}\{c_{ij}(a, b)\}$
    $\alpha := c_{ij}(a, S(i, a, j))$
    **if** $(c_i(a) = \bot)$ **and** $(\alpha > \bot)$ **then**
      $flag :=$ TRUE
    **end if**
    Project$(i, a, j, \alpha)$
  **end for**
  ProjectUnary$(i)$
  return $flag$

---

**Function 7** PruneVar$(i)$

**Input:** a variable $i$
**Output:** true if the domain of $i$ is changed, false otherwise
  $change :=$ FALSE
  **for** $a \in d_i$ s.t. $(c_i(a) \oplus c_\emptyset = \top)$ **do**
    $d_i := d_i - \{a\}$
    $change :=$ TRUE
  **end for**
  return $change$

---

**Procedure 8** $AC^*$ ($R$ is used in Procedure 10 and 11)

```
Input: a COP instance P
  while Q ≠ ∅ do
    j := pop(Q)
    for c_ij ∈ C do
      if FindSupportAC*(i, j) then
        R := R ∪ {i}
      end if
    end for
    for i ∈ X do
      if PruneVar(i) then
        Q := Q ∪ {i}
      end if
    end for
  end while
```

In order to integrate $DAC^*$ and $AC^*$ to obtain $FDAC^*$, we obtain full supports for variable $i$ on $c_{ij}$ while preserving supports for all values of $j$ on $c_{ij}$, which can be done by extending the minimum cost of $c_j$ to $c_{ij}$ and then projecting $c_{ij}$ down to $c_i$.

**Function 9** FindFullSupportAC*$(i, j)$

```
Input: variables i and j
Output: true if there is a full support for i from j, false otherwise
  flag := FALSE
  for a ∈ d_i s.t. c_ij(a, S(i, a, j)) ⊕ c_j(S(i, a, j)) > ⊥ do
    S(i, a, j) := arg min_{b∈d_j} {c_ij(a, b) ⊕ c_j(b)}
    P[a] := c_ij(a, S(i, a, j)) ⊕ c_j(S(i, a, j))
    if (P[a] > ⊥) ∩ (c_i(a) = ⊥) then
      flag := TRUE
    end if
  end for
  for b ∈ d_j do
    S(j, b, i) := arg max_{a∈d_i} {P[a] − c_ij(a, b)}
    E[b] := P[S(j, b, i)] − c_ij(a, b)
  end for
  for b ∈ d_j do
    Extend(j, b, i, E[b])
  end for
  for a ∈ d_i do
    Project(i, a, j, P[a])
  end for
  ProjectUnary(i)
  return flag
```

FindFullSupportAC*$(i, j)$ (Function 6) enforces supports for all values of variable $i$ on $c_{ij}$. It returns True whenever the cost of a value in $d_i$ has been increased to above $\perp$. $DAC^*$ enforces directional arc consistency by maintaining a global priority queue $R$ which stores all the variables whose costs of values have been increased from $\perp$, because some variables may lose their full supports and new supports need to be enforced on those. It also inserts a variable into the queue

**Procedure 10** $DAC^*$

```
Input: a COP instance P
  while R ≠ ∅ do
    j := pop(R)
    if PruneVar(j) then
      Q := Q ∪ {j}
    end if
    for each c_ij ∈ C s.t. i < j do
      if FindFullSupportAC*(i, j) then
        R := R ∪ {i}
      end if
    end for
  end while
  for each i ∈ X do
    if PruneVar(i) then
      Q := Q ∪ {i}
    end if
  end for
```

**Procedure 11** FDAC*. Initially, $Q = R = X$

```
Input: a COP instance P
  while (Q ≠ ∅) ∨ (R ≠ ∅) do
    AC*()
    DAC*()
  end while
```

$Q$, which stores variables who may have lost their supports, whenever a value is pruned. $DAC^*$ iterates until $R$ is empty. $FDAC^*$ is enforced by enforcing $AC^*$ and $DAC^*$ simultaneously; it iterates both $R$ and $Q$ until they are empty. The following theorems present the complexity of $DAC^*$ and $FDAC^*$.

**Theorem 3.2.** *(Larrosa and Schiex, 2003) The time complexity of* $AC^*$ *is* $O(n^2d^3)$ *and the space complexity is* $O(ed)$.

**Theorem 3.3.** *(Larrosa and Schiex, 2003) The time complexity of* $DAC^*$ *is* $O(ed^2)$ *and the space complexity is* $O(ed)$, $e$ *is the number of constraints, and* $d$ *is largest domain size.*

**Theorem 3.4.** *(Larrosa and Schiex, 2003) The complexity of* $FDAC^*$ *is time* $O(end^3)$ *and space* $O(ed)$, *where* $n$ *is the number of variables,* $e$ *is the number of constraints, and* $d$ *is the largest domain size.*

### 3.2.4 EDAC

While the soft arc consistency algorithm $FDAC^*$ produces a zero-arity constraint $c_\emptyset$ stronger than $DAC^*$ or $AC^*$ alone (Section 3.5), a better algorithm called existential arc consistency (EAC)

produces an even stronger zero-arity constraint than $FDAC^*$ (Givry and Zytnicki, 2005). They observed that in some problems that are already $FDAC^*$, the zero-arity constraint can be further increased by trying to find a support and full supports on at least one value of $i$ with respect to every constraint affecting $i$.

**Definition 3.9.** *(Givry and Zytnicki, 2005) Variable $x_i$ is existential arc consistent if there is at least one value $a \in d_i$ such that $c_i(a) = \bot$ and it has a full support in every constraint $c_{ij}$. A WCSP is existential arc consistent ($EAC^*$) if every variable is node and existential arc consistent.*

The arc consistencies $AC^*, DAC^*$, and $FDAC^*$ require every value of a domain to satisfy the same property. $EAC^*$ is different from those because it requires the existence of a special value of a domain. According to the definition of $EAC^*$, if a WCSP is not existential arc consistent, then for a variable $i$, every value $a$ s.t. $c_i(a) = \bot$ does not have a full support in at least one constraint $c_{ij}$, which means $\forall j$, $\forall b \in d_j, c_{ij}(a, b) \oplus c_j(b) > \bot$. Therefore enforcing full supports in $x_i$ will increase the cost of every value $a$ s.t. $c_i(a) = \bot$, thus enabling us to project the unary costs of $i$ down to $c_\emptyset$ to increase the lower bound. The following definition combines the benefits of $EAC^*$ and $FDAC^*$ to achieve a stronger local consistency property.

For example, Figure 3.5 shows a WCSP instance which is $FDAC^*$ but we can further increase $c_\emptyset$ by extending 1 cost from $c_y(b)$ to $c_{yz}$, projecting 1 cost from $c_{yz}$ down to $(z, a)$, extending 1 cost from $c_x(a)$ to $c_{xz}$, projecting 1 cost from $c_{xz}$ down to $(z, b)$, and projecting $c_z$ down to $c_\emptyset$. The result is shown in Figure 3.6.

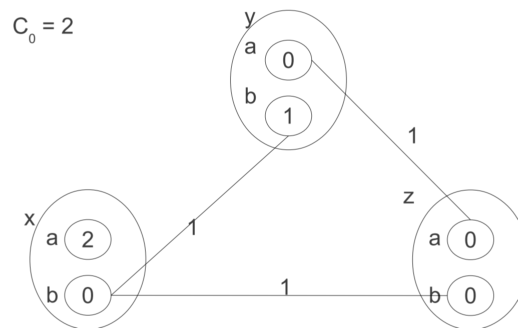



**Figure 3.5:** A WCSP instance that is $FDAC^*$ but not $EDAC^*$

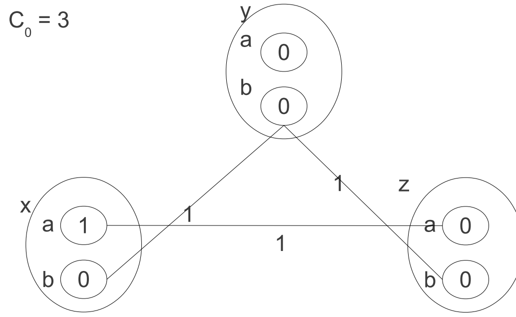



**Figure 3.6:** A WCSP instance that is $EDAC^*$

The previous example shows a way to increase $c_\emptyset$ after the problem is $FDAC^*$. This example illustrates another form of soft arc consistency called $EDAC^*$ which maintains a stronger $c_\emptyset$ than $FDAC^*$. Its definition is as follows.

**Definition 3.10.** *(Givry and Zytnicki, 2005) A WCSP is $EDAC^*$ if it is $FDAC^*$ and $EAC^*$.*

$EDAC^*$ requires that every value is fully supported in one direction and supported in the other direction in order to satisfy $FDAC^*$, and at least one value per variable must be fully supported in both directions in order to satisfy $EAC^*$. It can be shown that $EDAC^*$ reduces to classical arc consistency in CSPs.

The algorithm for enforcing $EDAC^*$ is presented in Algorithm 13. Besides the usual auxiliary functions introduced in Section 3.2.3 (FindSupports, FindFullSupports, PruneVar), it requires FindExistentialSupport($i$) (Function 12), which enforces the existential support in $i$ by finding full supports with respect to every lower adjacent variable $j$. As long as there is a variable $j$, which is lexicographically earlier than $i$ and does not have a full support for $i$, Fuction 12 keeps finding full supports in $j$ for $i$.

## 3.3 xAC

xAC generalizes the definition of arc consistency in classical CSPs (Horsch *et al.*, 2002). This generalization leads to useful application in classical CSPs as well as soft CSPs. In this section, we review the definition and relevant applications of xAC.

**Function 12** FindExistentialSupport($i$)

**Input:** a variable $i$
**Output:** true if there is an existential support the variable $i$, false otherwise

```
flag := FALSE
c_β = ⊕min_{b∈d_j}{c_ij(a,b) ⊕ c_j(b)}, c_ij ∈ C, j < i
α := min_{a∈d_i}{c_i(a) ⊕ c_β}
if α > ⊥ then
   for each c_ij ∈ C s.t. j < i do
      flag := FindFullSupports(i, j) ∨flag
   end for
end if
return flag
```

**Procedure 13** Enforcing $EDAC^*$. Initially, $Q = R = S = X$.

**Input:** a COP instance $P$

```
 1: while Q ≠ ∅ ∨ R ≠ ∅ ∨ S ≠ ∅ do
 2:    P := {l|i ∈ S, l > i, c_il ∈ C} ∪ S
 3:    S := ∅
 4:    while P ≠ ∅ do
 5:       i := popMin(P)
 6:       if FindExistentialSupport(i) then
 7:          R = R ∪ {i}
 8:          for each c_ij ∈ C s.t. j > i do
 9:             P := P ∪ {j}
10:          end for
11:       end if
12:    end while
13:    while R ≠ ∅ do
14:       j := popMax(R)
15:       for each c_ij ∈ C s.t. i < j do
16:          if FindFullSupports(i, j) then
17:             R := R ∪ {i}
18:             S := S ∪ {i}
19:          end if
20:       end for
21:    end while
22:    while Q ≠ ∅ do
23:       j := popMin(Q)
24:       for each c_ij ∈ C s.t. i > j do
25:          if FindSupports(i, j) then
26:             R := R ∪ {i}
27:             S := S ∪ {i}
28:          end if
29:       end for
30:    end while
31:    for each i ∈ X do
32:       if PruneVar(i) then
33:          Q := Q ∪ {i}
34:       end if
35:    end for
36: end while
```

In classical CSPs, arc consistency can be seen as a method to eliminate values which cannot be part of any solution. If the constraint graph is a tree, arc consistency can eliminate values if and only if those values do not appear in any solution of the problem. If the constraint graph is not a tree (i.e., the graph contains cycles), then arc consistency prunes values soundly, but not completely. In other words, AC can be seen as the approximation of a problem, namely finding the values in each domain which may be used in a solution. However, arc consistency can still be applied in general cases in order to obtain approximation of the solutions.

Based on the idea of arc consistency, xAC obtains marginals for all values of the variables, which indicate the values for potential solutions in classical CSPs and approximate the costs of values in COPs.

**Definition 3.11.** *(Horsch* et al.*, 2002) Let $P = \langle C, con \rangle$ be a semiring constraint problem with constraints $C$ and variables $con$. The marginal solution of $P$ on variables $I \subset con$, written $M_I(P)$ is defined as follows:*

$$\mathcal{M}_I(P) = (\otimes_{c \in C} c) \Downarrow_I$$

*When $I$ contains a single variable $X$, we write $\mathcal{M}_X = (\otimes_{c \in C} c) \Downarrow_{\{X\}}$.*

The above definition shows the foundation of xAC, which projects the solution of the SCSP onto a single variable. The operations $\otimes$ and $\Downarrow$ are defined using the semiring operators $\times$ and $+$ (see Definition 2.17).

Obviously, the definition of $\mathcal{M}_I$ is not practical for computation, since it requires the global solution $\otimes_{c \in C} c$. However, it is possible to calculate $\mathcal{M}_I$ in the special case of a tree structured constraint graph (Horsch *et al.*, 2002). The technique can be described as follows.

Let $P_x$ be the set of constraints in $P$ on the variable $x$ (i.e., $P_x = \{c \in C | x \in c.con\}$, where $c.con$ is defined in Definition 2.15). In a tree-structured constraint graph, removing any variable $x$ and the associated constraints on $x$ creates a number of independent, smaller constraint problems, $P_1, \cdots, P_m$, and $x$ shares exactly one constraint $c \in C$ with a single variable $n_i \in P_i, 1 \leq i \leq m$. If the marginal solutions $M_{n_i}(P_i)$ are known, then

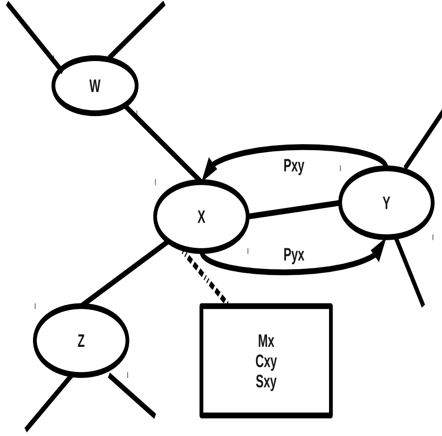



**Figure 3.7:** The parameters of an arbitrary node X

$$\mathcal{M}_x(P) = \otimes_{i=1}^{m}\{[c_i \times \mathcal{M}_{n_i}(P_i)] \Downarrow_x\}, \text{where } x \in c_i.con. \tag{3.1}$$

In other words, we can compute the marginal solution on $x$ if we know the marginal solutions to the subproblems containing $x$'s neighbours. If the problem is tree-structured, $M_x(P)$ is the exact marginal of variable $i$; if it is not tree-structured, then $M_x(P)$ approximates the marginal.

The xAC algorithm is an iterative technique for computing $\mathcal{M}_x(\mathcal{P})$ for every $x \in \mathcal{P}$. The technique can be described in terms of message passing. A message is composed of a vector of valuations, one for each element of the sending variable. Each variable $x$ stores the constraint $c_{xy}$, and receives messages consisting of a vector $P_{xy}$ from each neighbour $y$. Variable $x$ also computes a vector $S_{xy} = (c_{xy} \otimes P_{xy}) \Downarrow_x$, and can compute another vector $M_x = \bigotimes_y S_{xy}$, which has one element for each domain value of $x$. Outgoing messages to neighbour $y$ from variable $x$ are calculated by $P_{yx} = \bigotimes_{u \neq y} S_{xu}$; in other words, the outgoing messages from $x$ to $y$ consist of the marginal at $x$ leaving out any content that may have originated at $y$. Figure 3.7 shows the xAC parameters of an arbitrary node $X$ graphically.

In xAC, an iteration consists of the steps needed to update the marginal solution $M_x$ for each variable $x$. It can be shown that each iteration requires worst case $O(ed^2)$ time, and that by repeating the iteration, the marginals $M_x$ will converge in tree structured constraint graphs in $g$

iterations, where $g$ is the diameter of the graph. However, if the constraint graph contains cycles (i.e., not a tree), xAC may not converge and the iterations can run indefinitely. Therefore, in this thesis, we allow xAC to iterate at most 100 times and stop the propagation after 100 iterations no matter whether it converges or not. This technique is very similar to the local computations performed in constraint propagation in classical constraint problems, and also similar to Pearl's algorithm (Pearl, 1988) for calculating marginal probabilities in singly-connected Bayesian networks (Horsch and Havens, 2000).

However, the xAC algorithm can be used to approximate the marginal solutions in constraint problems of arbitrary structure, in much the same way that Pearl's algorithm can be used in multiply-connected Bayesian networks, and arc consistency can be applied to arbitrary classical CSPs (Murphy *et al.*, 1999). The approximation may converge, or it can be halted after a finite number of steps. The marginals are used as value ordering heuristics.

xAC allows the derivation of local propagation algorithms for any problems which have a join operation ($\otimes$) and a projection operation ($\Downarrow$). It is possible to derive implementations for MaxCSP, WCSP, and so on (Horsch *et al.*, 2002).

In this thesis we extended the basic implementation of xAC to introduce five new variations that use different amounts of propagation. These variations differ from each other in terms of how the xAC heuristic is utilized to guide the search.

- The first variation is called Vanilla, which uses the original version of xAC (Horsch *et al.*, 2002). This variation simply uses xAC as the value ordering heuristic to guide the search. Since there is no value pruning, the performance of this algorithm relies on the quality of heuristic; the earlier a good potential solution is found, the better the performance will be.

- The second variation is called TreePruning (TP). When the problem is tree-structured, xAC converges to exact marginals in only $d$ iterations, where $d$ is the diameter of the remaining graph. Since our implementation only propagates on the active constraints which involve at least one unassigned variable, we can detect tree-structured problems by observing that the difference in marginals is exactly zero in successive iterations. When this situation is detected,

an optimal solution for the subproblem can be obtained for the current node by assigning the best value based on the exact marginal of that variable. The cost of this solution is compared with the upper bound. If it is worse, we prune the branch, otherwise we continue as normal. This is similar to DAC (Dechter and Pearl, 1987).

- The third variation is called OnOff. The xAC procedure is very expensive in terms of constraint checks, so we limit the amount of propagation xAC does by turning it on and off in various situations. We assume the value ordering heuristic provided by xAC leads to a good solution, which can be used by a plain branch-and-bound search to effectively prune the branches without the need of xAC. Therefore, we turn off xAC whenever we find a better solution. When xAC is turned off, it does no propagation and the marginal solutions most recently computed while it was on are used for value ordering. However, whenever we reach a full assignment that is worse than the current best, the previous marginal solutions are no longer helpful and need to be recomputed, so we turn on xAC again. When xAC is turned on, it does full propagation at every node, and the value ordering heuristic is used as normal. This variation starts with xAC turned on.

- The fourth variation is called Radical Pruning (RP), which uses the value ordering to prune values at each new instantiation. After instantiating a new value for the next variable, RP deletes the value with the worst marginal valuation (thus the last in the value ordering heuristic) in the next variable. Because the heuristic provided by xAC is only an approximation of the true cost when it's not tree-structured, RP is not sound. However, if the approximation is good, it is unlikely that the best solution will use the deleted values. Notice that RP is an unsound and incomplete algorithm, because it may prune a branch which contains the true solution.

- The fifth variation is called TP+OnOff, which combines TP and OnOff.

- The sixth variation is called xAC*, which combines TP, OnOff, and RP.

These variations are evaluated in Chapters 4, 5, and 6.

## 3.4 Virtual Arc Consistency

In this section, we briefly review Virtual Arc Consistency (VAC), a state-of-the-art soft AC algorithm (Cooper *et al.*, 2010). VAC produces a stronger lower bound than $EDAC^*$ by planning sequences of *rational* soft AC equivalence transformation. VAC generalizes classical arc consistency, using the VCSP framework and costs which are modeled by rational numbers. VAC has multiple phases to analyze the state of the problem. It tries to find a sequence of SAC operations that will increase the $c_\emptyset$ more than $EDAC^*$. It will be described in more detail below.

In Section 3.2.4, the WCSPs were introduced using a valuation structure $S = (\mathcal{N} \cup \{+\infty\}, +, >)$, where $\mathcal{N}$ is the set of non-negative integers. Since VAC transfers rational costs across the whole constraint graph, a more general valuation structure which can deal with non-negative rational numbers is needed. The new valuation structure is $\mathcal{Q}^+ = (\mathcal{R}^+ \cup \{+\infty\}, +, \geq)$, where $\mathcal{R}^+$ is the set of non-negative rational numbers. Similarly, the valuation structure $S_m = (\{0, 1, \cdots, m\}, \oplus, \geq)$ can be embedded in $\mathcal{Q}^+_m = (\mathcal{R}^+_m \cup \{\infty\}, \oplus, \geq)$, where $\mathcal{R}^+_m$ is the set of rational numbers $\alpha$ s.t. $0 \leq \alpha < m$.

**Definition 3.12.** *(Cooper* et al.*, 2010) If $P = \langle X, D, C, S, \phi \rangle$ is a VCSP over the valuation structure $\mathcal{Q}^+$ or $\mathcal{Q}^+_m$, then Bool(P) is the classical CSP $\langle X, D, \bar{C} \rangle$ where, for all scopes $S \neq \emptyset$, $\langle S, R_S \rangle \in \bar{C}$ if and only if $\exists \langle S, c_S \rangle \in C$, where $R_S$ is the relation defined by the following property: $\forall x \in l(S), t \in R_S \Leftrightarrow c_S(t) = 0$.*

A CSP is *empty* if at least one of its domains is empty.

**Definition 3.13.** *(Cooper* et al.*, 2010) A VCSP P is virtual arc consistent if Bool(P) is arc consistent.*

The following theorem shows that it is always possible to increase the lower bound $c_\emptyset$ if Bool($P$) is inconsistent during AC enforcement.

**Theorem 3.5.** *(Cooper* et al.*, 2010) Let P be a VCSP over the valuation structure $\mathcal{Q}^+$ or $\mathcal{Q}^+_m$ such that $c_\emptyset < \infty$. Then there exists a sequence of soft AC operations which when applied to P*

*leads to an increase in $c_\emptyset$ if and only if the arc consistency closure of Bool(P) is empty.*

VAC is stronger than $EAC^*$ introduced in section 3.2.4. $EAC^*$ can be seen as applying a single iteration of VAC. In $EAC^*$, weights are transferred virtually to each variable from all its neighbours.

**Corollary 3.1.** *(Cooper* et al.*, 2010) If a VCSP P over the valuation structure $\mathcal{Q}^+$ or $\mathcal{Q}^+_m$ is VAC, then establishing EDAC cannot increase the lower bound $c_\emptyset$ in P.*

### 3.4.1 Propagating VAC

The VAC propagation consists of three phases (Cooper *et al.*, 2010):

- Instrumented-AC phase: In this phase, a $Bool(P)$ is constructed from the original $P$ and standard classical AC is propagated over this $Bool(P)$ (see Definition 3.12). If a domain becomes empty, i.e., the last remaining value in the domain is deleted, the deleted variable which is referred to as the wiped-out variable, is recorded as well as the reasons for deleting any values during the propagation. This phase collects data to construct a path which will be used in a later phase. This path is called a R-path which is a first-in-first-out queue consisting of SAC operations.

- Computing $\lambda$: In this phase, VAC computes the R-path which lead to the value deletions of the domain in $Bool(P)$. The R-path is used by VAC to find a maximum allowable cost $\lambda$ in the original $P$.

- Applying equivalence-preserving transformations: In this phase, a sequence of soft AC operations, projections and extensions, is applied to the original $P$ following the R-path and transferring the cost $\lambda$ along the path. Eventually, all the values of the wiped-out variable in the Instrumented-AC phase will have at least $\lambda$ cost each, which means this $\lambda$ cost can be unary-projected from the wiped-out variable down to $c_\emptyset$.

Overall, the space complexity of the algorithm is $O(erd)$ where $e$ is the number of constraints, $r$ is the maximum arity of cost functions, and $d$ is the largest domain size. The time complexity of

one iteration of the algorithm is $O(ed^r)$.

It can be shown that VAC can enter an infinite loop when solving some problems by increasing the lower bound by a smaller and smaller $\lambda$ (Cooper *et al.*, 2010). To address this situation, the authors introduced a heuristic to VAC: if a certain number of iterations never improve the lower bound by more than a threshold $\epsilon$, the VAC propagation will stop and the branch-and-bound search continues.

This heuristic version of VAC is called $VAC_\epsilon$ (Cooper *et al.*, 2010). If a valuation structure $Q_m$ is used, the maximum allowable cost is $m$, which means the number of iterations is at most $O(\frac{m}{\epsilon})$. Therefore, the complexity of consistency propagation for a binary constraint optimization problem is $O(\frac{ed^2m}{\epsilon})$.

## 3.5 Search

A hybrid CSP or COP algorithm usually consists of consistency propagation and search. Since there are four different soft AC algorithms being considered, namely W-AC*2001, EDAC, xAC, and VAC, and variations of these ACs, we required a common search template so that the consistency propagation would be decoupled from the search and we could insert different soft AC algorithms into the same search template. A search template which abstracts *arc consistency during search* (ACS) into a separate module decoupled from the depth first search in solving CSPs was introduced (Likitvivatanavong *et al.*, 2007). We adapted and modified this template to solve COPs. We call our branch-and-bound search template Maintaining Soft Arc Consistency (MSAC).

The essence of MSAC consists of one data field $P$ which is a COP, and three methods which are try($x = a$), backjump($x = a$), and addInfer($x \neq a$), where $x \in P.V$, and $a \in P.D_x$. The symbols $P.V$, $P.D_x(x \in P.V)$, and $P.C$ denote the set of variables, the domain of $x$, and the set of constraints in $P$.

The problem $P$ is encapsulated inside MSAC. The method try($x = a$) enforces arc consistency on $P \cup \{x = a\}$. If it is arc consistent, then try($x = a$) sets $P$ to $P \cup \{x = a\}$ and returns true, otherwise $P$ remains the same, $\{x = a\}$ is discarded, and try($x = a$) returns false. The method

addInfer($x \neq a$) enforces arc consistency on $P.C \cup \{x \neq a\}$. If the problem is arc consistent, addInfer($x \neq a$) sets $P.C$ to $P.C \cup \{x \neq a\}$ and returns true, otherwise it returns false. The method backjump($x = a$) retracts the effects of all try()s and addInfer()s since the inclusion of $x = a$. All the changes made to the domains and constraints are discarded and $P$ returns to the state before $x = a$ was included and an inequality $x \neq a$ was added to $P$.

We adapted the search template ACS into Algorithm 14 which maintains soft arc consistency during branch-and-bound search. Any soft arc consistency can be chosen to propagate new assignments or inequalities to the whole problem. Line 17 retrieves the lower bound of a soft AC algorithm and compare it against the current upper bound; if the lower bound is smaller than the upper bound, we continue to search the subproblem belong the current partial assignment, otherwise we backtrack to previous states. The lower bound is an estimation of the cost of the current partial assignment and the upper bound is the cost of the best solution found so far.

Algorithm 15 calls a SAC algorithm to propagate the assignment $(x, a)$ to the whole problem. Algorithm 16 propagates an inequality $(x, a)$ to $P$. Algorithm 17 restores $P$ back to a previous state before $x = a$. SAC.propagate($P$) returns false when some variable's domain becomes empty during propagation.

## 3.6 Summary

In this chapter, we reviewed several soft AC algorithms, including W-AC*2001, EDAC, VAC, and xAC. There is no comparison of W-AC*2001, EDAC, or VAC against xAC in the literature. Currently, VAC is the best soft AC algorithm for solving large COPs (Cooper *et al.*, 2010).

**Procedure 14** Maintaining Soft Arc Consistency During Search (MSAC)

**Input:** $P = \{V, D, C\}$
**Output:** Solutions

```
1: create an empty stack S to store current partial assignment
2: freevariables ← P.V
3: assignments ← ∅
4: keepsearching ← true
5: while keepsearching do
6:   if freevariables = ∅ then
7:     if S.cost ≤ ub then
8:       solutions.add(S)
9:     else
10:      repeat
11:        (x, a) ← S.pop()
12:        backjump(x = a)
13:      until backjump(x = a) does not cause empty domains
14:    end if
15:  else
16:    select a variable x_i from freevariables and a value a for x_i
17:    if try(x_i = a) and SAC.lb < ub then
18:      S.push((x_i, a))
19:      freevariables ← freevariables −{x_i}
20:    else
21:      repeat
22:        if S is not empty then
23:          (x_i, a) ← S.pop()
24:          backjump(x_i = a)
25:          freevariables ← freevariables ∪{x_i}
26:        else
27:          keepsearching = false
28:          break
29:        end if
30:      until addInfer(x_i ≠ a) and backjump(x_i = a) does not cause empty domains
31:    end if
32:  end if
33: end while
34: return solutions
```

**Procedure 15** Procedure try($x = a$)

**Input:** a variable $x$ and a value $a \in d_x$
**Output:** the propagation result of a Soft AC algorithm

```
1: backup the internal data structures of SAC, following timestamp(x = a)
2: backup the current domains of P, following timestamp(x = a)
3: delete all values except a from d_x
4: return SAC.propagate(P)
```

**Procedure 16** Procedure addInfer($x \neq a$)

**Input:** a variable $x$ and a value $a \in d_x$
**Output:** the propagation result of a Soft AC algorithm

```
1: backjump(x, a)
2: delete a from d_x
3: return SAC.propagate(P)
```

**Procedure 17** Procedure backjump($x = a$)

**Input:** a variable $x$ and a value $a \in d_x$

```
1: restore the internal data structures of SAC, before timestamp(x = a)
2: restore the domains of P, before timestamp(x = a)
3: delete a from d_x
```

# Chapter 4

# Comparing the Value Ordering Heuristics of xAC, W-AC*2001, and EDAC

## 4.1 Purpose

W-AC*2001 and EDAC enforce arc consistency (AC) by shifting costs in the problem and deleting values that are inconsistent. After the deletion, unary constraints (i.e., the costs associated with the allowable values) of the variables are used as a plain value ordering in the branch-and-bound search. xAC, on the other hand, does not delete values but generates a value ordering based on the information inferred from the constraint problem.

A value ordering heuristic is crucial to a search procedure in solving COPs. With a value ordering of high quality, a branch-and-bound search procedure is likely to find a good solution (i.e., a solution with a low cost) early, thus enabling the branch-and-bound search to use the cost of that solution as the upper bound to prune subproblems, which leads to a smaller search space than a higher upper bound produced by a value ordering of lower quality. This section shows how well xAC performs as a value ordering heuristic against W-AC*2001 and EDAC.

In this section, we use W-AC*2001, EDAC, and xAC to propagate arc consistency on binary COPs, and compare the inferred value orderings against the exact one. The exact value ordering algorithm performs an exhaustive search over the whole search tree, producing an optimal value ordering which sets the best possible upper limit for a heuristic. Since generating the exact value ordering is exponential in time complexity, we use the algorithms to solve relatively small problem instances. We demonstrate that xAC resembles the exact ordering more than W-AC*2001 and

**Figure 4.1:** Two Binary Constraints in A Skewed COP Instance

| $c_{ij}$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |

| $c_{kl}$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |

EDAC. We expect that xAC should have a higher chance to produce a smaller search space than W-AC*2001 and EDAC.

## 4.2 Problem Instances

To evaluate the relative performance of the methods, we would like to use randomly generated problems. However, the standard problem generators tend to create problem instances with uniformly distributed costs. This makes heuristic information hard to compute, since every value instantiation choice is roughly the same. In order to create interesting problem instances so that not all value orderings are equally good, we generalize Skewed CSPs (Horsch *et al.*, 2002) into Skewed COPs to test the algorithms. Skewed CSPs are based on the "flawed" random CSP model (Gent *et al.*, 2001). In Skewed CSPs, the disallowed pairs are chosen from a smaller subset of possible pairs in about one third of the tuples in the constraints. More specifically, Skewed COPs use $\top$, the worst valuation, to replace disallowed paris and $\perp$, the best valuation, to replace allowed pairs. Figure 4.1 shows two binary constraints in a typical Skewed COP. For instance, the constraint between variable $x_0$ and $x_2$ specifies that the tuple $t = (2, 1)$ is associated with a cost of 0 where the third value is assigned to variable $x_0$ and the second value is assigned to variable $x_2$.

We generated random Skewed CSPs based on $\langle n, m, p_1, p_2 \rangle$, where $n$ is the number of variables, $m$ is the domain size, $p_1$ is the density of constraint graph, $p_2$ is the tightness of constraints. The problem set includes four classes, each of which corresponds to six hundred instances generated by the parameters $\langle n, 5, p_1, 0.5 \rangle$. Each class has a fixed $p_1$ and a varied $n$ ranging from 5 to 10. The $p_1$ of these four classes is assigned from one value in $\{0.3, 0.5, 0.7, 0.9\}$, respectively.

## 4.3 Methods

We propagate AC on these problems using W-AC*2001, EDAC, xAC, and an exact value ordering algorithm to calculate a node cost matrix, respectively. A node cost matrix $N$ is a $n \times m$ matrix where each element $N_{i,j}$ represents a cost associated with the $jth$ value of the $ith$ variable.

We use the following statistical variables to calculate the correlation between the exact node cost matrix and the approximate one computed by the heuristic algorithms.

**Definition 4.1.** *(Kendall, 1990) Given two vectors $X = (x_1, x_2, \cdots, x_n)$ and $Y = (y_1, y_2, \cdots, y_n)$, the Spearman's rank correlation coefficient is computed using the following formula:*

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{4.1}$$

*where, $d_i = x_i - y_i$ is the difference between the ordered ranks of corresponding values $x_i$ and $y_i$, and $n$ is the number of values in each data set.*

We used the Spearman's rank correlation coefficient (SRCC) to indicate the degree of ranking correlation between the exact value ordering and the heuristics. We did not use the Pearson's correlation coefficient (PCC), because, as a value ordering heuristic, the order in which the values are attempted is more important than the accuracy of the cost estimates resulting from propagation.

For example, suppose we have a node cost matrix for a COP of two variables, each of which has two values. Suppose the node cost matrix for the exact value ordering is $\left(\begin{smallmatrix} 3 & 5 \\ 2 & 1 \end{smallmatrix}\right)$ and the node cost matrix for the xAC value ordering is $\left(\begin{smallmatrix} 1 & 2 \\ 5 & 2 \end{smallmatrix}\right)$. Then, we compute the rank-order matrices, which contain rankings of values based on their costs; the lower the cost, the higher the ranking. For example, the first value of the first variable is ranked higher than the second value of the first variable in the exact value ordering. Then we get the rank-order matrix for the exact value ordering, $\left(\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}\right)$, and the rank order matrix for the xAC, $\left(\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}\right)$.

The heuristic algorithms propagate on these problems without any search, while the exact algorithm solves the whole problem completely by trying every possible value combination. Although the exact algorithm generates the best value ordering, it is computationally impractical for large

problems. Therefore, we only use the best value ordering as a reference against which the heuristic algorithms perform.

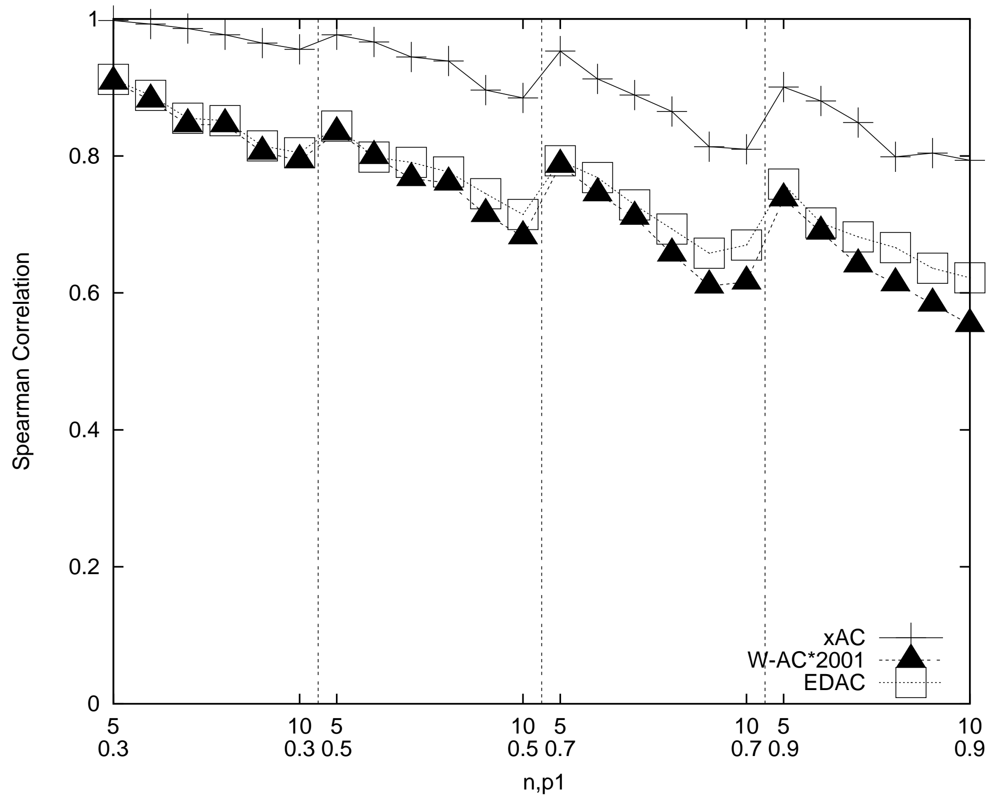
## 4.4 Empirical Results

Figure 4.2 shows the comparison of the aforementioned value orderings generated by xAC, W-AC*2001, and EDAC, respectively. This graph combined results from four separate experiment groups. On the $X$-axis, $n$ is the number of variables and $p_1$ is the density of the constraint graph. A smaller $p_1$ means fewer constraints. On the $Y$-axis, the Spearman's rank correlation score is calculated using the $\rho$-value (Definition 4.1), which represents a degree of resemblance between the value ordering generated by an algorithm and the exact best value ordering generated by a brute-force algorithm. A higher Spearman correlation indicates a value ordering of higher quality. In each experiment group, we fix $p_1$ and generate problem instances with different numbers of variables from 5 to 10, respectively. For each number of variable, we run the algorithms on 50 random instances and calculate the average of the $\rho$-value for these algorithms. We do not include error bars in the comparison because the standard deviation of the $\rho$-values is so small that they will not be visible on the graph. The $X$-axis represents experiments which are divided into four groups. Each group has a fixed constraint graph density $p_1$, which ranges from 0.3 to 0.9 across all groups. Within each group, the number of variables $n$ ranges from 5 to 10. The result shows that xAC generates better value ordering than both W-AC*2001 and EDAC, and EDAC is slightly better than W-AC*2001.

By extracting and combining information of costs from the whole constraint graph through message passing, xAC is able to produce value orderings of higher quality, especially when the width of the graph is small. W-AC*2001 and EDAC, on the other hand, do not produce a value ordering with comparable quality, because they gather only local information by maintaining either supports or full supports in the neighbours of variables while xAC iteratively gathers information from the whole constraint graph.

**Figure 4.2:** Quality Comparison of Value Ordering Heuristics

The $x$-axis is labelled by 2 parameters: $n$ ranging from 5 to 10 and $p_1$ ranging from 0.3 to 0.9



# Chapter 5

# Empirical Comparison of Soft AC Algorithms in Solving Random COPs

In this chapter, we use random COP instances to compare the performance of xAC (the variants TP and TP+OnOff described in Section 3.3), EDAC, and VAC. Section 5.2 compares the algorithms in solving randomly generated Max CSPs (MaxCSPs). Section 5.3 compares the algorithms in solving randomly generated Uniform Integer COPs (UICOPs).

## 5.1 Preliminary

We compare the algorithms in terms of their time complexities measured by the number of constraint checks and the number of nodes traversed. In our work, the following steps count as a constraint check:

- checking the valuation of a tuple in a binary constraint;
- checking the valuation of a value in a unary constraint;
- changing the valuation of a tuple or a value due to a projection, an extension, or a unary projection.

A node in a branch-and-bound search represents a single value assignment during the search. A node is traversed when a value is assigned to a variable in Algorithm 14. The number of nodes is the number of node traverses during the whole branch-and-bound search.

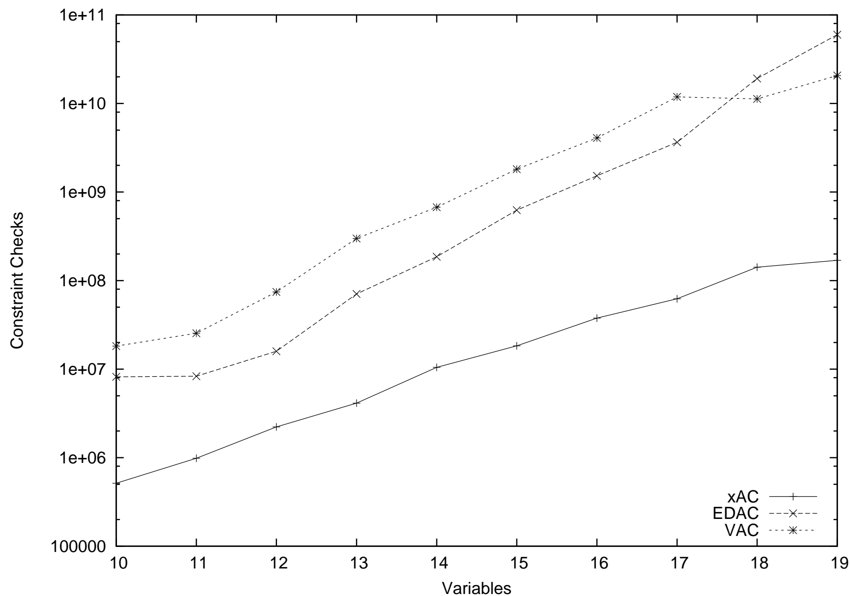



**Figure 5.1:** Comparing Number of Constraint Checks on MaxCSPs

## 5.2 MaxCSPs

We used a Skewed CSP (Horsch *et al.*, 2002) generator to produce random problem instances. By specifying 1 as the maximum allowable cost in any constraint tuple, we have generated random MaxCSP problems. In MaxCSPs, each allowed tuple has zero cost, and each forbidden tuple has one cost.

Each instance is generated using the parameters $\langle n, m, p_1, p_2 \rangle$, where $n$ is the number of variables, $m$ is the domain size, $p_1$ is the density of the constraint graph, and $p_2$ is the tightness of the constraint. These problem instances includes ten clusters, each of which corresponds to 50 random instances generated by the parameters $\langle n, 5, 0.5, 0.5 \rangle$, where $n$ is chosen from $\{10, 11, \cdots, 19\}$. The parameters were chosen to keep runtime reasonably small and the problem instances away from extreme values.

Figure 5.1 shows the number of constraint checks performed by the soft AC algorithms. We present the results on a logarithmic scale. Clearly, xAC performs fewer constraint checks than EDAC and VAC. VAC performed more constraint checks than EDAC until the number of variables

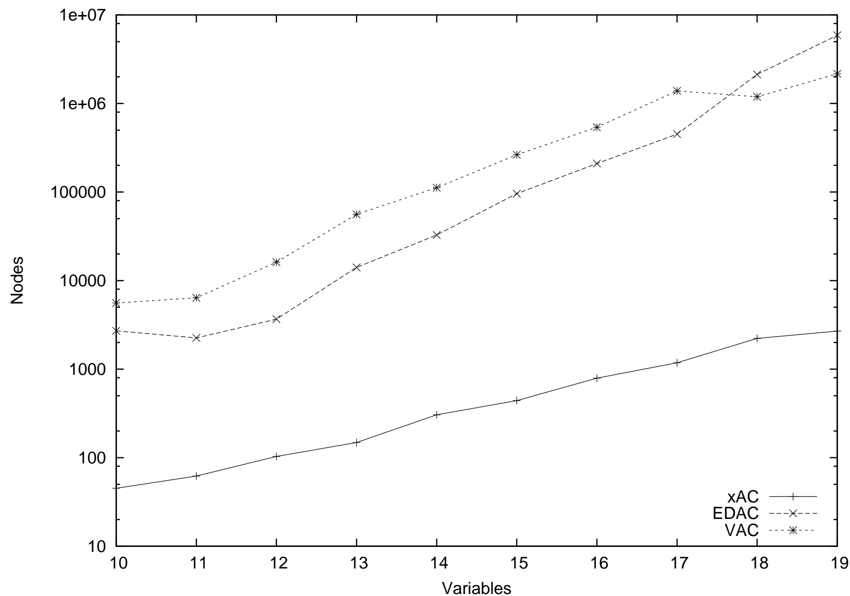



**Figure 5.2:** Comparing Number of Nodes on MaxCSPs

reaches 18. The confidence intervals of the data points from EDAC and VAC do not overlap when the number of variables is smaller than 18 while they overlap when the number of variables is 18 and 19. For this reason, we don't plot them in Figure 5.1 and 5.2.

Figure 5.2 shows the number of nodes searched by the soft AC algorithms. Obviously, xAC searched fewer nodes than both EDAC and VAC. VAC searched more nodes than EDAC until the number of variables reaches 18.

Although this empirical result demonstrated the efficiency of xAC, the problem instances are small. One of the reasons why VAC performed worse than xAC include the high cost of maintaining complex data structures during the propagation. Another reason is that, although VAC is able to handle a special type of constraints called sub-modular constraints very well (Cooper *et al.*, 2008), the proportion of such sub-modular constraints is low compared to other problem instances such as RLFAPs (see Section 6.2). A sub-modular binary constraint can be defined as the summation of generalized interval (GI) functions (Cohen *et al.*, 2004). A GI function maps a domain value to a fixed cost. A sub-modular binary constraint can be constructed by summing several GI functions together, where the domain values in the GI functions are uniformly sampled. Sub-

modular constraints are applied in finite-valued and boolean domains to represent a tractable constraint problem class.

## 5.3 Uniform Integer COPs

Unlike MaxCSPs, whose tuples of constraints take costs of 0 or 1, the constraint tuples of Uniform Integer COPs (UICOPs) take integer costs from a range. By including more choices of costs for the tuples, we can create COPs that are not easy to model as MaxCSPs and compare the algorithms on a different kind of problems. Our UICOP instances include costs randomly chosen from the set $\{0, 1, \cdots, 10\}$.

Figure 5.3 shows the number of constraint checks performed by the soft AC algorithms. We present the results on a logarithmic scale. xAC performs more constraint checks than EDAC and VAC when the number of variables is below 16 and fewer constraint checks when it is larger than or equal to 16. In addition, the number of constraint checks performed by xAC and VAC grows slower than EDAC once the number of variables is larger than 15. When the number of variables is 19, most of the problem instances need more than 4 hours to solve. Since we tested the algorithms on 50 problem instances, we did not try the algorithms on larger problems. The confidence intervals of the data points do not overlap when the number of variables is smaller than 14 while they overlap when the number of variables is between 14 and 19. For this reason, we don't plot them in Figure 5.3 and 5.4.

Figure 5.4 shows the number of nodes searched by the soft AC algorithms. The shape of this graph is similar to Figure 5.3, since an increase in the number of nodes searched entails an increase in the number of constraint checks, and vice versa. The algorithm xAC searches more nodes than EDAC and VAC when the number of variables is below 14, but it searches fewer nodes than VAC when the number of variables is larger than or equal to 14. The number of nodes searched by xAC grows slower than EDAC and VAC once the number of variables is larger than 14.

Compared with MaxCSPs, the experimental results of UICOPs show an increase of the number of constraint checks performed and nodes searched by xAC and VAC. One possible reason might

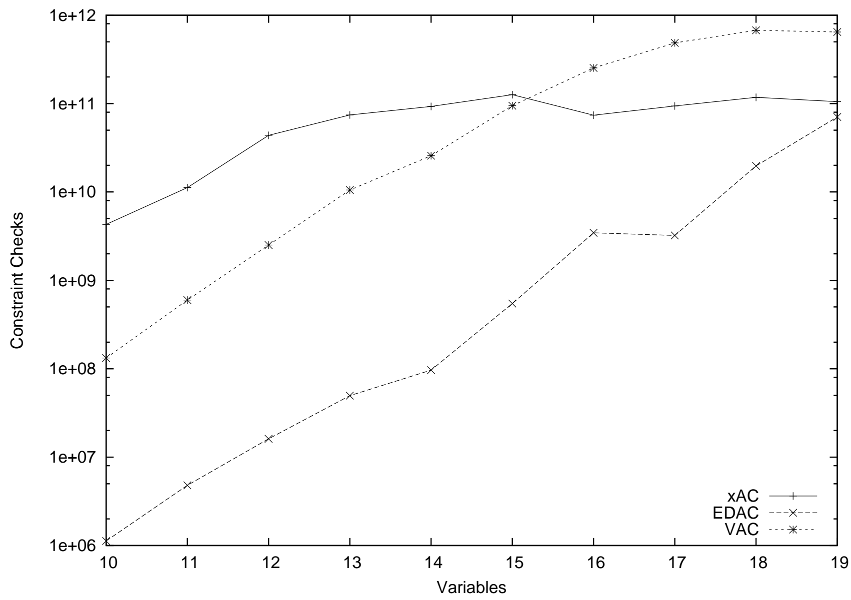



**Figure 5.3:** Comparing Number of Constraint Checks on UICOPs

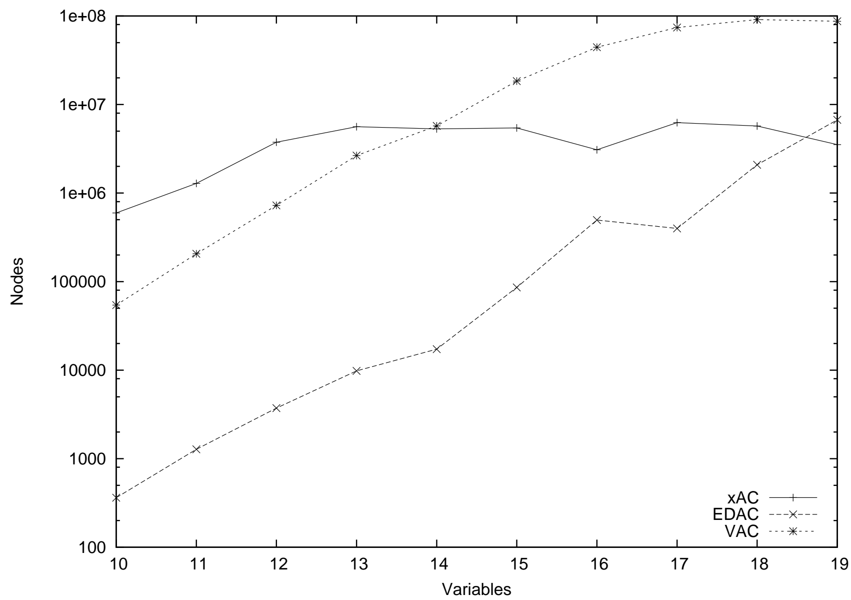



**Figure 5.4:** Comparing Number of Nodes on UICOPs

be that the variety of constraint costs increases the difficulty for the algorithms to converge, which leads to more work performed by these algorithms during consistency propagation.

In the range of 14-19 variables, the results show that the number of constraint checks performed and the number of nodes searched by xAC and VAC increase more slowly than EDAC. It may look like these two trends for xAC and VAC level out near 17-19 variables, but we think this trend is local and not indicative of a greater trend. Another reason why the performance seems to level out might be that the performance of xAC on the problems with 10-13 variables is worse than VAC's performance on these problems. In addition, the performance of xAC on the problems with 10-13 variables is relatively worse than xAC's performance on the problems with 17-19 variables in terms of work performed per variable. We expect these two trends to increase exponentially for larger problem instances. Although Figure 5.3 and 5.4 cannot be treated as a comprehensive comparison of xAC, EDAC, and VAC, they show that xAC was able to efficiently solve some UICOPs.

## 5.4 Summary

In this chapter, we compared the performance of xAC, EDAC, and VAC on MaxCSPs and UICOPs. For MaxCSPs, xAC performed fewer constraint checks and searched fewer nodes than EDAC and VAC. For UICOPs, xAC performed more constraint checks and searched more nodes than EDAC and VAC when the number of variables was small. But it performed fewer constraint checks and searched fewer nodes than VAC when the number of variables became larger. In addition, the number of constraint checks and nodes of xAC seems to grow slower than EDAC and VAC.

The variety in the constraint costs seems to affect the performance of the algorithms. For MaxCSPs, a constraint cost is taken from the set $\{0, 1\}$. For UICOPs, a constraint cost is taken uniformly from the set $\{0, 1, \cdots, 10\}$. The increased number of choices for the costs seems to decrease the chance of convergence for both xAC and VAC, which seems to increase the work performed by these algorithms. In addition, the value ordering generated by xAC for UICOPs seems to be worse than that of MaxCSPs, which might be caused by the increased variety of constraint costs as well.

The performance comparison of xAC, EDAC, and VAC solving randomly generated COPs can help a person to decide which algorithm should be used to solve which real world problem. For example, if the structure of a real world problem resembles the structure of MaxCSPs, we expect xAC to perform better than EDAC and VAC. On the other hand, if the structure of a real world problem resembles the structure of UICOPs, we cannot predict which algorithm would be better than others. In this case, these algorithms will be used to solve the problem and the empirical comparison of these algorithms can decide which one is the most suitable one for the problem.

# Chapter 6

# Empirical Comparison of xAC, EDAC and VAC in Solving Real-World COPs

In this chapter, we compare the number of constraint checks and nodes of xAC, EDAC and VAC on solving real world problems. We test the algorithms to solve the benchmarks *Uncapacitated Warehouse Location Problems* (Givry and Zytnicki, 2005) and *Radio Link Frequency Assignment Problems* (Cooper *et al.*, 2010). Then, we convert the problem instances of *Quasigroup Problems* (Gomes and Shmoys, 2002) into COPs and test the algorithms to solve these problems.

In this chapter, we measure the time complexities of the algorithms in terms of the number of constraint checks performed and the number of nodes searched. These two measurements are described in Section 5.1. The algorithms are tested on a machine with a 2.4GHz Intel Core 2 Quad CPU and 8 GB of RAM.

## 6.1 Uncapacitated Warehouse Location Problems

There are two variants of the warehouse location problem, uncapacitated and capacitated. This section describes the Constraint Optimization model for the **Uncapacitated Warehouse Location Problem** (UWLP) (Givry and Zytnicki, 2005). In UWLPs, the problem models the scenario faced by a super market chain opening warehouses at some locations in order to supply its existing stores. The objective is to decide which warehouses should be opened, and which store should be supplied by which warehouse, such that the sum of the storage and supply costs is minimized. Each store must be supplied by exactly one open warehouse. The formal definition of UWLP is described in the following paragraphs (Kratica *et al.*, 1996).

Suppose there are $n$ warehouses and $m$ stores. Let $t_i$ $(1 \leq i \leq n)$ be a storage cost in a warehouse $i$, and $h_{ij}(1 \leq i \leq n, 1 \leq j \leq m)$ a cost of shipment from warehouse $i$ to store $j$. The objective is to choose open warehouses and a shipment plan from warehouses to stores, where the total cost is minimized.

Let $x_i(x_i \in \{0,1\}, i = 1, \cdots, n)$ denote the *ith* element of a boolean array indicating that the warehouse $i$ is open or not. If $x_i = 0$, then the warehouse is closed. If $x_i = 1$, then it is open. Let $y_{ij}$ be the quantity of shipment from warehouse $i$ to store $j$. Without loss of generality, the $y_{ij}$ of each store is normalized to 1 (Kratica *et al.*, 1996). The goal is to find a value assignment to all the $x_i$ and $y_{ij}$ that minimizes

$$\sum_{i=1}^{n} t_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{m} h_{ij} y_{ij}.$$

subject to the following conditions:

$$\sum_{i=1}^{n} y_{ij} = 1, \qquad j = 1, 2, \cdots, m;$$

$$0 \leq y_{ij} \leq x_i \text{ and } x_i \in \{0,1\}, \qquad i = 1, 2, \cdots, n; j = 1, 2, \cdots, m.$$

### 6.1.1 WCSP Formulation of UWLP

The problem can be modelled by $n$ boolean variables for the warehouses indicating whether a warehouse is open or closed, $m$ integer variables for the stores with $n$ domain values each of which is the identifier of a warehouse, $n + m$ soft unary constraints for the shipment costs, and $n \times m$ hard binary constraints indicating which warehouses supply which stores.

There are two types of costs: storage costs and shipment costs. Storage costs are naturally modelled as unary constraints. Although we can model shipment costs as binary constraints, we have to take into account the fact that each store is supplied by exactly one open warehouse, which is implicitly specified in the goal of the problem. Therefore, we model shipment costs as unary constraints for the store variables, and binary constraints for the one-to-one mapping between open warehouses and stores. The formal description is as follows.

Let $L = \{l_1, l_2, \cdots, l_n\}$ represent $n$ candidate warehouses, each of which has two values $\{0, 1\}$, where 0 means the warehouse is closed and 1 means it is open. Let $S = \{s_1, s_2, \cdots, s_m\}$ represent

**Table 6.1:** Storage Costs

| $l_1$ | $l_2$ | $l_3$ |
|---|---|---|
| 10 | 20 | 15 |

$m$ stores, each of which has $n$ values $\{1, 2, \cdots, n\}$ for the supplying warehouses. For each $l_i$, the unary constraint is:

$$c_i = \begin{cases} t_i & \text{if } l_i = 1 \text{, where } t_i \text{ is the storage cost in warehouse } i \\ 0 & \text{if } l_i = 0. \end{cases}$$

For each $s_i$, the unary constraint is:

$$c_i(j) = h_{ij}.$$

Notice that we use unary constraints to represent shipment costs from warehouses to stores. In addition, there is a binary constraint $w_{ij}$ between every pair of store and warehouse:

$$c_{ij} = \begin{cases} k & \text{if } s_i = j \wedge l_j = 0 \\ 0 & \text{otherwise} \end{cases}$$

where $k$ is an intolerable cost forbidding a closed warehouse supplying any store. We assigned the maximum allowable integer cost in Java to $k$ in our implementation.

### 6.1.2 Examples

In this section, we present a small example of UWLP using the two formulations mentioned in the previous section. Suppose we have three warehouses and two stores. The first formulation includes 3 warehouse variables $\{l_1, l_2, l_3\}$ and 2 store variables $\{s_1, s_2\}$. The storage costs are presented in Table 6.1.

Without loss of generality and for the sake of simplicity, we assume $y_{ij} = 1, \forall i, j$. Therefore, each store only needs 1 unit of shipment from the warehouses. The shipment costs are presented in Table 6.2.

The unary constraints for the warehouse variables are presented in Table 6.3.

**Table 6.2:** Shipment Cost

| | $l_1$ | $l_2$ | $l_3$ |
|---|---|---|---|
| $s_1$ | 30 | 15 | 45 |
| $s_2$ | 10 | 25 | 15 |

**Table 6.3:** Unary Constraints for Warehouses

| | 0 | 1 |
|---|---|---|
| $c_1$ | 0 | 10 |
| $c_2$ | 0 | 20 |
| $c_3$ | 0 | 15 |

**Table 6.4:** Unary Constraints for Stores

| | 1 | 2 | 3 |
|---|---|---|---|
| $c_4$ | 30 | 15 | 45 |
| $c_5$ | 10 | 25 | 15 |

**Figure 6.1:** Binary Constraints

| $c_{11}$ | 0 | 1 |
|---|---|---|
| 1 | k | 0 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |

| $c_{12}$ | 0 | 1 |
|---|---|---|
| 1 | 0 | 0 |
| 2 | k | 0 |
| 3 | 0 | 0 |

| $c_{13}$ | 0 | 1 |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 0 | 0 |
| 3 | k | 0 |

| $c_{21}$ | 0 | 1 |
|---|---|---|
| 1 | k | 0 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |

| $c_{22}$ | 0 | 1 |
|---|---|---|
| 1 | 0 | 0 |
| 2 | k | 0 |
| 3 | 0 | 0 |

| $c_{23}$ | 0 | 1 |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 0 | 0 |
| 3 | k | 0 |

**Table 6.5:** UWLP instances

| | #Variables | #WCSP Constraints | Max Domain Size |
|---|---|---|---|
| cap51 | 66 | 63 | 16 |
| cap61 | 66 | 63 | 16 |
| cap62 | 66 | 63 | 16 |
| cap63 | 66 | 63 | 16 |
| cap64 | 66 | 63 | 16 |
| cap71 | 66 | 63 | 16 |
| cap72 | 66 | 63 | 16 |
| cap73 | 66 | 63 | 16 |
| cap74 | 66 | 63 | 16 |

| | #Variables | #WCSP Constraints | Max Domain Size |
|---|---|---|---|
| cap81 | 75 | 99 | 25 |
| cap82 | 75 | 99 | 25 |
| cap84 | 75 | 99 | 25 |
| cap101 | 75 | 99 | 25 |
| cap102 | 75 | 99 | 25 |
| cap103 | 75 | 99 | 25 |
| cap104 | 75 | 99 | 25 |
| capmo1 | 200 | 495 | 100 |
| capmo2 | 200 | 435 | 100 |

The unary constraints for store variables are presented in Table 6.4.

In binary constraints, we use $k$ to represent an intolerable cost. Whenever a warehouse $j$ is closed, i.e., $x_j = 0$, we forbid the situation where a store is supplied by $j$ by assigning the tuple $(s_i = j, x_j = 0)$ a cost of $k$. A binary constraint $c_{ij}$ models a relationship between a store $s_i$ and a warehouse $l_j$. The binary constraints are presented in Figure 6.1. In this figure, $c_{ij}$ represents a binary constraint between a store $s_i$ and a warehouse $l_j$, where the row headers are values in the domains of warehouse variables and the column headers are values in the domains of store variables. A cell $(a, b)$ in a $c_{ij}$ table represents the cost associated with assigning value $a$ to the store variable $s_i$ and value $b$ to the warehouse variable $l_j$.

**Table 6.6:** Number of Constraint Checks on UWLPs

| | TP | TP+OnOff | EDAC |
|---|---|---|---|
| cap51 | 1.24E+07 | 1.86E+07 | - |
| cap61 | 2.30E+07 | 1.86E+07 | - |
| cap62 | 1.05E+07 | 2.33E+07 | 11.93E+07 |
| cap63 | 1.24E+07 | 1.86E+07 | 11.94E+07 |
| cap64 | 1.31E+07 | 1.86E+07 | - |
| cap71 | 2.30E+07 | 1.86E+07 | - |
| cap72 | 1.05E+07 | 2.33E+07 | 11.93E+07 |
| cap73 | 1.24E+07 | 1.86E+07 | 11.94E+07 |
| cap74 | 1.31E+07 | 1.86E+07 | 44.77E+07 |

| | TP | TP+OnOff | EDAC |
|---|---|---|---|
| cap81 | - | 8.16E+07 | - |
| cap82 | 3.46E+07 | 9.16E+07 | - |
| cap84 | 1.29E+08 | 8.64E+07 | - |
| cap101 | - | 8.16E+07 | 44.77E+07 |
| cap102 | 3.46E+07 | 9.16E+07 | - |
| cap103 | 9.24E+07 | 8.33E+07 | - |
| cap104 | 1.29E+08 | 8.64E+07 | - |
| capmo1 | 3.52E+09 | 1.15E+10 | - |
| capmo2 | 3.77E+09 | 1.16E+10 | - |
| capmo3 | 3.51E+09 | 1.16E+10 | - |
| capmo4 | 3.51E+09 | 1.16E+10 | - |

### 6.1.3 Empirical Results

We use the benchmark problems (Heras *et al.*, 2005) to compare $EDAC^*$ against TP and TP+OnOff. For each problem instance, we set up a time limit of 5 hours to control the total runtime of the experiment. If an algorithm cannot solve a problem instance within the time limit, we use "-" to represent the timeout in the experimental results. We did not include other variations of xAC and VAC because they could not solve any of the problem instances within the time limit. The xAC variation TP performs better than the original version because it finds a solution once the sub-problem becomes a tree while the original keeps searching, therefore TP avoids unnecessary search and consistency propagation performed by the original (see Section 3.3).

The experimental results in Table 6.6 show that TP and Tp+OnOff perform one order of magnitude fewer constraint checks than EDAC. In addition, TP and TP+OnOff are comparable to each other.

Our implementations of EDAC and VAC exceeded the time limit on some of the problem instances. However, these algorithms were able to solve these problems in other people's implemen-

tation (Cooper *et al.*, 2010). The difference lies in implementation detail. Since we are using MSAC (see Section 3.5), we feel that the comparison is fair in a relative sense, even if our implementation is slower than the original. The reasons for the performance difference between our implementation and the original could be:

- The Java implementation is not as efficient as C++. EDAC has been implemented in C++ which is able to solve cap71 to cap104 (Heras *et al.*, 2005).

- We did not implement a last conflict driven variable selection heuristic, variable elimination during search and dichotomic branching (Cooper *et al.*, 2008).

Nevertheless, for all the problems solved in the given amount of time, the number of constraint checks performed is in favor of xAC. This shows xAC performs less work in a branch-and-bound search with a lexicographical variable ordering.

## 6.2 Radio Link Frequency Assignment Problems

This section presents the Radio Link Frequency Assignment Problems (RLFAPs) and experimental results of performance comparison. RLFAPs aim at assigning limited spectral resources to a set of links, and keeping the interference between the links to a minimum level. This problem is NP-hard (Cabon *et al.*, 1999).

A real world benchmark has been built by CELAR (the french "Centre d'Electronique de l'Armement"), who gathered data from a real network. It is publicly available as the project CALMA (Combinatorial Algorithms for Military Applications; see http://www.win.tue.nl/ wscor/calma.html).

These problems are very interesting for potential Constraint Programming research: the problems can be represented by unary and binary constraints and the domain sizes are finite. These problems are used as a benchmark due to the enormous number of variables and constraints.

### 6.2.1 Informal Description

When radio communication links, i.e., connections between radio transmitters, are assigned the same or closely related frequencies, they may interfere with each other. Assume we have a radio communication network, including a set of radio links and a set of frequencies which come from limited spectral resources. To solve a RLFAP is to assign a frequency to each of these links such that the links can work simultaneously with minimum interference. In addition, the assignment has to obey certain physical constraints of the transmitters. If the previous constraints are satisfied, then we will prefer an assignment that uses as few spectral resources as possible, because we may want to use the remaining resources to extend the current network. Finally, when several bands are available, the lower bands are usually preferred to higher bands for reasons such as radio wave propagation and ease of deployment.

In most cases, the number of available frequencies is much smaller than the number of links. In the CELAR problems, the largest number of frequencies is 48 and the largest number of links is 916.

Finally, we may face two kinds of problems. The first is bulk assignments, where all transmitters and frequencies will be assigned. The second is updating assignments, where a subset of the transmitters may have pre-assigned frequencies. When we are dealing with the second kind, we should not modify the transmitters with pre-assigned frequencies as much as possible. So, combining the aforementioned constraints, we can see that a real-world RLFAP aims at finding an assignment which keeps the interference to a minimum level, spares resources by using as few frequencies as possible, uses low frequencies as much as possible, and maintains pre-assigned frequencies as much as possible (Cabon *et al.*, 1999).

In this thesis, several original benchmark WCSP instances are parsed and solved by our constraint solver. The original benchmarks represent various real-world scenarios including minimization of interference and maintenance of pre-assigned frequencies (see Appendix). We used these problem instances to experiment with the algorithms.

### 6.2.2 Formal Definition of the CELAR Problems

Suppose there is a set of radio links, called $X$. For each link $i \in X$, a frequency $f_i$ has to be chosen from a finite set $d_i$ of frequencies which generate unary constraints

$$f_i \in d_i. \tag{6.1}$$

Another way to describe the unary constraints is to model them as domains. For each link $i \in X$, the frequency $f_i$ being assigned to it comes from a domain $d_i$ of frequencies available for the transmitter.

In the updating problem, some links may have a pre-assigned frequency which define unary constraints

$$f_i = p_i. \tag{6.2}$$

Binary constraints are defined on pairs of links $\{i, j\}$. A constraint may be either an interference

$$|f_i - f_j| > d_{ij}, \tag{6.3}$$

where $d_{ij}$ is the minimum distance required between frequency $i$ and $j$, or a duplex

$$|f_i - f_j| = \delta_{ij}, \tag{6.4}$$

where $\delta_{ij}$ is defined by technological constraints on transmitters and frequency $i$ and $j$ must be exactly $\delta_{ij}$ away from each other.

The situation modeled by equation 6.3 allows interference between the two links $i$ and $j$. Usually, there are two types of interference: co-site interference and far-site interference. Co-site interference happens when two transmitters are close. Far-site interference happens when two transmitters are separated by a distance that is larger than ideal.

In equation 6.4, a duplex link is defined by $i$ and $j$. A duplex link is a connection between two sites $site_i$ and $site_j$, where one link is used to communicate information from $site_i$ to $site_j$ while the other is used for $site_j$ to $site_i$. The distance $\delta_{ij}$ is defined by specification constraints on transmitters, such as stable transmitting distance and most efficient transmitting mode.

Constraints described in equations 6.1 and 6.4 are always classical (i.e., true or false). In addition to the hard constraints, some constraints may be soft which we can violate at certain cost. For example, we can change pre-assigned frequencies with a mobility cost $m_i$, and we can assign $f_i, f_j$ such that $|f_i - f_j| \leq d_{ij}$ which violates the soft constraints described in equation 6.3 with an interference cost $c_i$. The complete set of constraints $C$ is therefore partitioned into a set $H$ of hard constraints and a set $S$ of soft constraints. However, this partition is irrelevant in the WCSP model which is shown later.

Although there are various optimization criteria, several problems for solving RLFAP are defined as follows (Cabon *et al.*, 1999):

- Feasibility (FEAS): the problem is to find an assignment of frequencies to each link such that all constraints in $C$ are satisfied. This problem acts like a preliminary test before a harder problem variant is approached. Notice that the feasibility version of the RLFAP is NP-complete, because we can use the constraints described in equations 6.1 and 6.3 to express any $k$-colouring problem, which is NP-complete (Angelsmark, 2005).

- Minimum span (SPAN): if all the constraints in $C$ can be satisfied simultaneously, we can try to minimize the largest frequency. The SPAN problem is more complex than the FEAS problem by a constant factor: "it can be reduced to a short sequence of FEAS problems using dichotomic search (and can simply be cast as a Possibilistic/Fuzzy CSP by adding soft unary constraints on the domain values)" (Cabon *et al.*, 1999).

- Minimum cardinality (CARD): if all the constraints in $C$ can be satisfied, we can try to minimize the number of different frequencies. CARD is harder than SPAN: it is not easy to express the problem in terms of binary constraints, which means we need to introduce $k$-arity constraints ($k > 2$) (Schiex *et al.*, 1995).

- Maximum Feasibility (MAX): if not all the constraints in $C$ can be satisfied, we can try to satisfy all "hard" constraints and minimize the sum of all the violation costs for "soft" constraints. In this thesis, we cast the original problems to SCSP-based WCSPs, which are

the default input for the xAC solver. The WCSP can be equivalently transformed into VCSP which is the input format for the EDAC and VAC solver.

### 6.2.3 WCSP Formulation of RLFAP

A weighted CSP (WCSP) is a tuple $(X, D, C, W, m)$ (Cooper *et al.*, 2008). $X$ and $D$ are sets of $n$ variables and domains, respectively. The domain of variable $i \in X$ is denoted $d_i$. For a set of variables $S \subset X$, we use $t(S)$ to represent the set of all possible tuples over the domains of variables of $S$. A cost function $w_S$ assigns integer costs to assignments of the variables in $S$, i.e., $w_S$: $t(S) \rightarrow [0, m]$. The set of possible costs is $[0, m]$ and $m$ is a large integer which represents an intolerable cost. $C$ is the set of constraints, each of which contains the scope $S$ and the cost function $w_S$.

For RLFAP, the set $X$ of unidirectional radio links is the set of variables. Each link $i \in X$ is a variable and its value is chosen from $d_i$, which is a finite set of frequencies available for that radio link. $C$ is a set of binary constraints. For each $c_j \in C$, the scope $S_j$ includes two variables. We use $S_j$ to represent variables associated with a constraint instead of *con* (see Definition 2.15), because we want to conform our notation with the literature (Cooper *et al.*, 2010). The cost function $w_S$ is defined in the following way:

- If $c_i$ with a scope $S_i$ represents an interference (Equation 6.3), we enumerate each tuple $t = (t_1, t_2) \in t(S_i)$ and calculate the absolute value of the difference $d^{'}_{ij} = |t_1 - t_2|$. If $d_{ij} > d^{'}_{ij}$, we assign 0 cost to $t$, otherwise we assign a violation cost to $t$, or we assign the intolerable cost $m$ to $t$ if no violation cost is specified. After we assign a cost to each tuple $t \in t(S)$, we get a two dimensional table representing the soft constraint with each cell containing the cost corresponding to each tuple, i.e., the possible assignment of the two variables.

- If $c_i$ represents a duplex link (Equation 6.4), we enumerate each tuple $t = (t_1, t_2) \in t(S)$ and calculate the absolute value of the difference $d^{'}_{ij} = |t_1 - t_2|$. If $d^{'}_{ij} = \delta_{ij}$, we assign 0 cost to $t$, otherwise we assign the intolerable cost $m$ to $t$.

| | #Variables | #WCSP Constraints |
|---|---|---|
| CELAR6-SUB0 | 16 | 31 |
| CELAR6-SUB1 | 14 | 43 |
| CELAR6-SUB2 | 16 | 59 |
| CELAR6-SUB3 | 18 | 69 |
| CELAR7-SUB0 | 16 | 29 |
| CELAR7-SUB1 | 14 | 32 |
| CELAR7-SUB2 | 16 | 44 |
| CELAR7-SUB3 | 18 | 68 |

| | #Variables | #WCSP Constraints |
|---|---|---|
| graph05 | 100 | 121 |
| graph06 | 200 | 270 |
| graph07 | 141 | 124 |
| graph11 | 304 | 463 |
| graph12 | 252 | 253 |
| graph13 | 458 | 557 |
| scen06 | 100 | 175 |
| scen07 | 200 | 413 |
| scen09 | 200 | 163 |

**Table 6.7:** RLFAP Instances

Due to the available time we have on this project, we did not convert the Minimum Span (SPAN) and Minimum Cardinality (CARD) problems (Section 6.2.2) in the RLFAP problem pool into WCSPs. However, we convert the FEAS or MAX problems into WCSPs. A time limit of 5 hours is given to each algorithm per problem instance.

### 6.2.4 Experimental Results

The properties (i.e., number of variables and number of constraints) of each problem instance is given in Table 6.7. The maximum domain size of all problem instances is 44.

We again allowed each algorithm to take at most 5 hours to solve one problem instance. The number of constraint checks performed is shown in Table 6.8. TP+OnOff is able to solve more problem instances than TP, because turning off the xAC propagation when it's not needed can save unnecessary constraint checks.

The number of nodes searched is presented in Table 6.9. For the solved problem instances, TP+OnOff searched more nodes because turning off the xAC propagation leads to less accurate value ordering which in turn may lead the branch-and-bound algorithm to search more nodes before reaching a complete assignment. One might notice that the number of nodes searched by TP and

| | TP | TP+OnOff |
|---|---|---|
| CELAR6-SUB0 | - | 5.72E+08 |
| CELAR6-SUB1 | 1.50E+08 | 8.51E+07 |
| CELAR6-SUB2 | 2.65E+08 | 3.23E+09 |
| CELAR6-SUB3 | 3.28E+08 | 4.78E+09 |
| CELAR7-SUB0 | 4.40E+09 | 1.55E+09 |
| CELAR7-SUB1 | - | 2.14E+09 |
| CELAR7-SUB2 | 3.06E+08 | 3.25E+09 |
| CELAR7-SUB3 | - | 3.62E+09 |

| | TP | TP+OnOff |
|---|---|---|
| graph05 | 5.74E+09 | 6.39E+10 |
| graph06 | - | 1.22E+09 |
| graph07 | - | 7.46E+10 |
| graph11 | - | - |
| graph12 | - | 2.49E+11 |
| graph13 | 2.32E+10 | - |
| scen06 | 1.57E+09 | 7.02E+10 |
| scen07 | - | 2.85E+11 |
| scen09 | - | 6.51E+10 |

**Table 6.8:** Number of Constraint Checks for Solving RLFAPs

| | TP | TP+OnOff |
|---|---|---|
| CELAR6-SUB0 | - | 227 |
| CELAR6-SUB1 | 89 | 104 |
| CELAR6-SUB2 | 126 | 557 |
| CELAR6-SUB3 | 135 | 680 |
| CELAR7-SUB0 | 2992 | 393 |
| CELAR7-SUB1 | - | 452 |
| CELAR7-SUB2 | 140 | 568 |
| CELAR7-SUB3 | - | 539 |

| | TP | TP+OnOff |
|---|---|---|
| graph05 | 581 | 2915 |
| graph06 | - | 274 |
| graph07 | - | 3507 |
| graph11 | - | - |
| graph12 | - | 6611 |
| graph13 | 1263 | - |
| scen06 | 306 | 3145 |
| scen07 | - | 5573 |
| scen09 | - | 3071 |

**Table 6.9:** Number of Nodes Searched for Solving RLFAPs

TP+OnOff seems to be low, considering the size of the problem instances. There are two reasons for this result. The first reason is that the number of constraints in each problem is not large compared to the number of all possible constraints. In other words, the density of the constraint graph $p_1$ is low. For example, the constraint density $p_1$ of CELAR6-SUB1 is 0.47. The second reason is that there are highly connected nodes in each problem instance, and assigning values to these variables first will lead to a tree-structured sub-problem relatively early in the search. For example, in the problem instance CELAR0-SUB1, there are three variables each of which has at least nine neighbours. If a partial assignment assigns values to these three variables, the search tree below that partial assignment is likely to be a tree-structured problem, which can be efficiently processed by TP and TP+OnOff (Section 3.3).

(a) Complete

| | | |
|---|---|---|
| B | A | C |
| A | C | B |
| C | B | A |

(b) Partial

| | | |
|---|---|---|
| B | | C |
| A | C | |
| | B | A |

**Table 6.10:** Quasigroup Examples

## 6.3 Quasigroup Problems

### 6.3.1 Problem Description

The Quasigroup Completion Problem (QCP) is a CSP benchmark (Gomes and Selman, 1997). Many real world problems, such as timetabling and routing, can be modelled as Quasigroups.

A quasigroup is a discrete structure whose multiplication table corresponds to a Latin Square. A Latin Square of order $n$ is an $n \times n$ table in which each one of $n$ distinct symbols occurs exactly once in each row and column. A partial quasigroup (or Latin Square) is a quasigroup (or Latin Square) in which some cells are empty. The Quasigroup Completion Problem (QCP) is formulated as follows: Given a partial quasigroup or order $n$, is there a solution to complete it?

Table 6.9(a) and 6.9(b) show a complete quasigroup and a partial one. Any kind of symbols can be used to construct a quasigroup.

Formally, a quasigroup is an ordered pair $(Q, \cdot)$, where $Q$ is a set and $\cdot$ is a binary operation on $Q$ such that the equations $a \cdot x = b$ and $y \cdot a = b$ are uniquely solvable for every pair of elements $a, b$ in $Q$. The $orderN$ of the quasigroup is the cardinality of the set $Q$. The equations are uniquely solvable for each pair $a, b$ in $Q$, therefore there exists unique elements $x$ and $y$ in $Q$ satisfying the equations.

We use a variant of the quasigroup problems called the Quasigroup With Holes (QWH) Problems to evaluate the algorithms. In QWHs, a complete random quasigroup is generated and symbols at random are removed to create a partial quasigroup. The goal is to complete the partial quasigroup by filling the empty cells. Both QCP and QWH problem instances are significantly harder when the holes are uniformly spread all over the table (Gomes and Shmoys, 2002). Unlike partial QCPs which may not have a solution, QWHs guarantee the existence of at least one solution. Since we know there will be at least one solution in any QWH problem, we stop the branch-and-bound search as soon as a full assignment with a cost of zero is found. In addition, as more and more symbols at random are removed, QWHs enter a phase transition region in which the generated problems are harder because they contain only one solution and many variables with large domains, thus increasing the difficulty to find the solution (Barták, 2006).

### 6.3.2 Problem Generations and Encodings

Although generating randomly distributed problems seem like a trivial task, it is in fact a complex process, where an ergodic Markov chain whose stationary distribution is uniform over a $N$ by $N$ grid is simulated (Jacobson and Matthews, 1996).

Given a partial Latin square of order $n$, PLS, a CSP $P = \langle X, D, C \rangle$ can be formulated as:

$$\begin{aligned} X &= \{x_{i,j} | x_{i,j} \in \{1, \cdots, n\}\}, \forall i, j; \\ D &= \{1, \cdots, n\}; \\ C_1 &= \text{alldiff}(x_{i,1}, x_{i,2}, \cdots, x_{i,n}\}, \forall i = 1, 2, \cdots, n; \\ C_2 &= \text{alldiff}(x_{1,j}, x_{2,j}, \cdots, x_{n,j}\}, \forall j = 1, 2, \cdots, n. \end{aligned}$$

The alldiff constraints indicate all the variables involved in the constraint must have different values. We convert an $n$-arity alldiff constraint into a set of $\frac{n \times (n-1)}{2}$ binary constraints.

### 6.3.3 Experimental Results

This section presents experimental results on the performance comparison of xAC, EDAC, VAC, and a plain branch-and-bound with random value ordering as a baseline to compare EDAC and

VAC for their performance on solving QWHs. The randomly generated problems are divided into 16 groups, each of which includes 50 instances. The order of the QWHs is $5 \times 5$ and the number of holes ranges from 5 to 20. Although there are 25 cells in the problem, the number of holes determine the number of significant variables in the problem.

There are two WCSP forms for the QWHs. The natural form converts each cell into a variable. For each of the holes, the corresponding variable has a unary constraint with $n$ zero costs. For each of the preassigned cells, the corresponding variable has a unary constraint with $n$ values where the value preassigned to the variable has zero cost and each of the remaining values has one cost. The preprocessed form takes out all preassigned cells by performing Forward Checking and leaves a constraint graph with variables for the holes. For each of the holes, the corresponding variable has $n$ values with a unary constraint of $n$ zero costs.

We converted the QWHs, which are CSPs, into the natural form by assigning one cost to false tuples and zero otherwise. Since every equivalent MaxCSP instance has at least one solution whose cost is zero, we stop the search as soon as a solution with zero cost is found. Each data point in the following graphs is the average of 50 instances and the value is presented on a logarithmic scale with base 10. We use BnB to represent the baseline algorithm which is a plain branch-and-bound with random value ordering in the experimental result figures.

Figure 6.2 shows the number of constraint checks performed when solving QWH instances. xAC performed more constraint checks than EDAC and VAC when the number of holes is small. When the number of holes increases, the number of constraint checks performed by xAC did not increase as fast as EDAC and VAC. This is because:

- xAC performed a lot of propagation when the number of holes is below 8. In other words, it performed a lot of work even though the problem is simple and a solution is easy to find.
- During xAC's propagation, the iterative process may not converge and the unary constraints of preassigned cells may change due to the possible existence of cyclic subgraphs. Although other algorithms always chose the correct value for preassigned cells, xAC may try incorrect ones for them which leads to larger than necessary search space.

- When the number of holes is larger than or equal to 8, xAC performed less propagation to find a good enough solution early because the value ordering heuristics gathered through global problem structure were better than EDAC and VAC. In addition, the fewer number of preassigned cells reduces the negative impact of xAC making incorrect choices for them. Unlike problems with smaller number of holes, larger number of holes increases the difficulty for other algorithms to make the correct choices for the majority of the variables since the number of preassigned variables decreases. Compared to the plain branch-and-bound, which uses a random value ordering, EDAC and VAC perform more constraint checks without reducing the number of nodes searched significantly (Figure 6.3).

The plain branch-and-bound without any consistency propagation performed the least number of constraint checks when the number of holes is low. This is because:

- The majority of checks happened inside a consistency propagation.

- Without any consistency propagation, a plain branch-and-bound searched more nodes than others but the problems were small enough for it to use a random value ordering to find a solution quickly.

Figure 6.3 showed the number of nodes searched when solving QWH instances. The plain branch-and-bound searched more nodes than others because no consistency propagation is available to provide any good value ordering. xAC searched more nodes than EDAC and VAC when the number of holes is below 16 but fewer when the number of holes is larger than or equal to 16 because:

- The WCSP instances specify a variable for each cell in the Quasigroup. Removing values from the variables with $n$ possible values can significantly reduce the search space. But xAC does not remove values like EDAC and VAC. When the number of holes is small, the problems are so simple that others can remove values from the variables to effectively prune the search space, while xAC still needs to search a larger space.

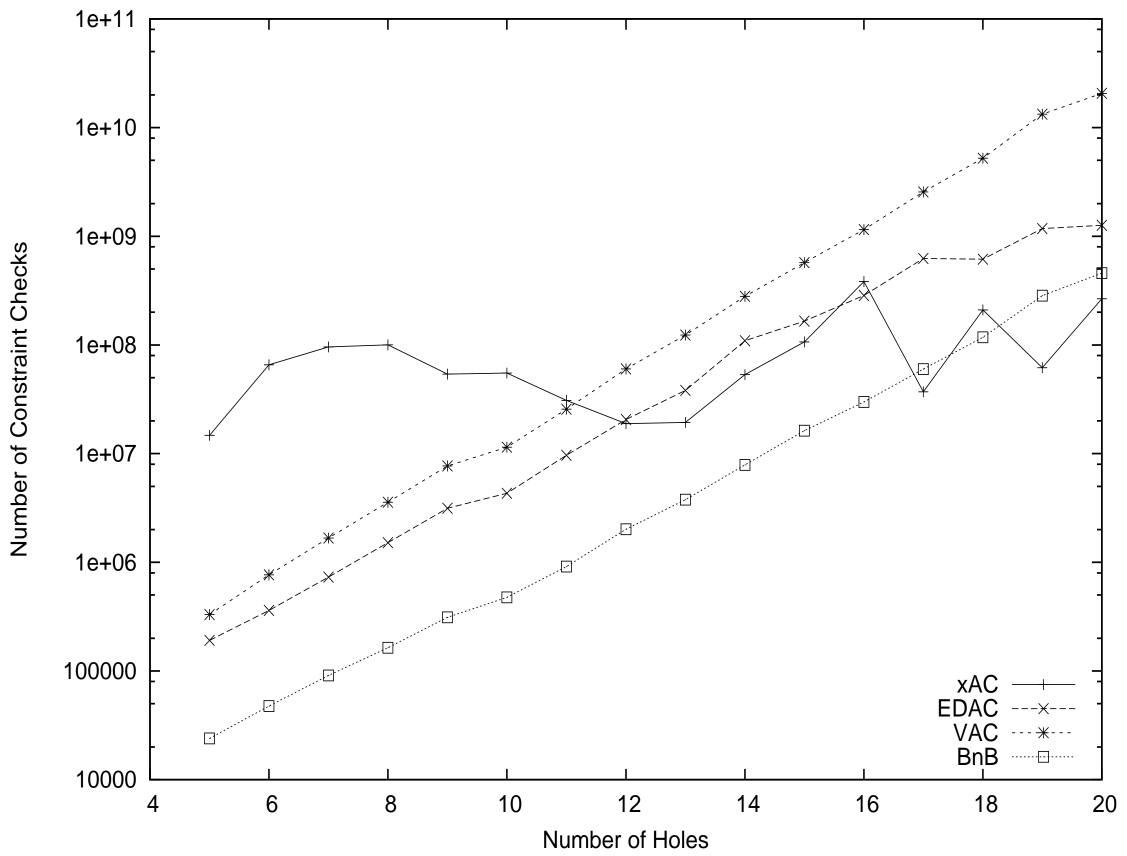



**Figure 6.2:** Number of Constraint Checks Performed for Solving Quasigroups

- xAC produced a good value ordering when the number of holes is high or when the number of solutions is high. A good value ordering leads to a good enough solution early in the search tree, which leads to effective pruning. In addition, for EDAC and VAC, a larger number of holes, i.e., a larger number of variables with $n$ domain values, reduces the effect of value pruning on the search space.

- When the problem size is small, xAC's propagation may iterate over a cycle in the constraint graph, which changes the unary constraints of preassigned cells to incorrect ones. Because the unary constraints of preassigned cells can be modified, the number of variables of all problem instances is the same for xAC. In other words, every problem instance has 25 variables. Then xAC makes unnecessary and wrong choices for those cells, which leads to more nodes being searched. However, when the problem size becomes larger, EDAC and VAC produce poor value ordering because a lot of the suboptimal solutions have close costs, while xAC is able to collect cost information from the whole constraint graph to calculate a good value ordering heuristic to guide the search.

Since the natural form of QWHs include preassigned cells, xAC may make incorrect choices for them. But in other problems (Chapter 4 and 5) where no variable is preassigned before the search starts, it may provide more dynamic and informed choices than other algorithms. Compared with other problems, the natural form of QWHs is special because a preprocessed form can be constructed from it by taking out the preassigned cells using forward checking. If xAC propagates consistency on a preprocessed QWH, we expect the number of constraint checks and nodes to decrease for the problems with small number of holes. If xAC has to propagate over a QWH in the natural form, we can increase xAC's performance on small problems by preventing it from changing the unary constraints of preassigned cells.

VAC searched more nodes than others when the number of holes is larger than or equal to 11. The possible reasons might include:

- VAC performed more consistency propagation than the other two because it tries to find a better lower bound. However, since the hard problem instances of QWHs usually have only

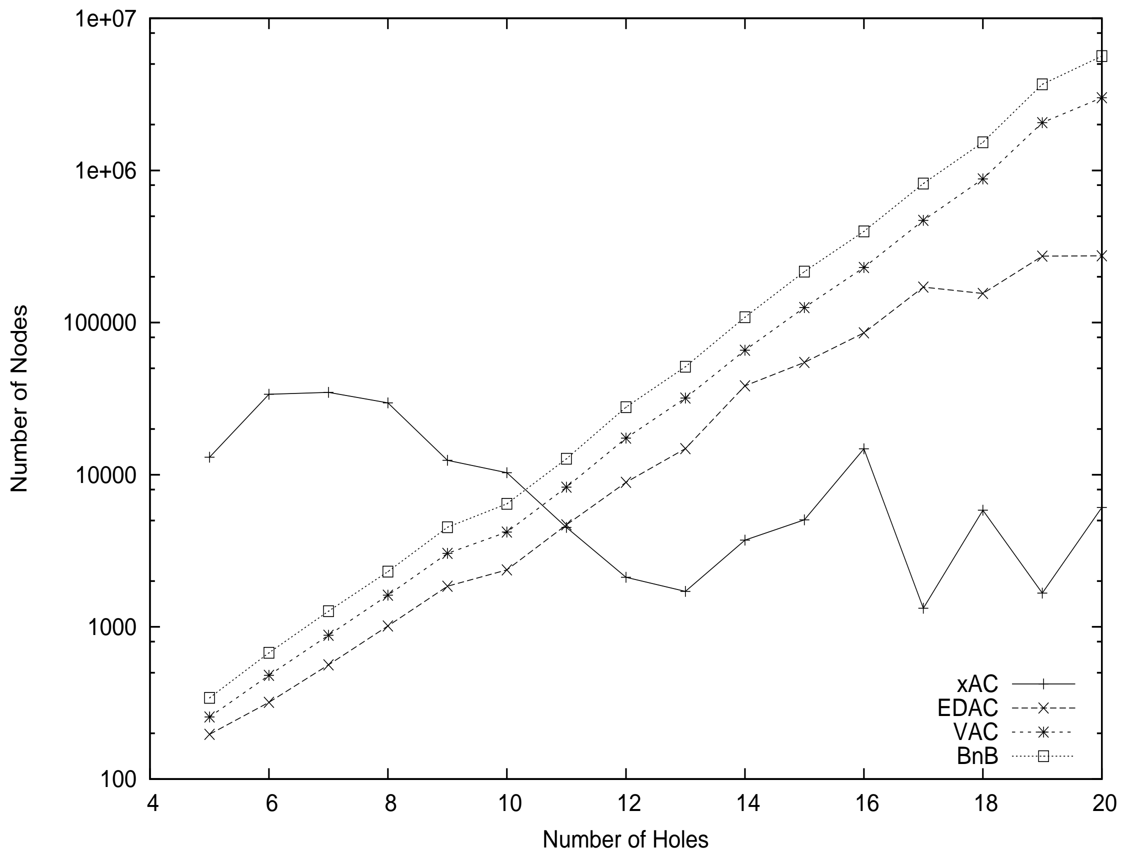



**Figure 6.3:** Number of Nodes Searched for Solving Quasigroups

one solution with zero cost and most of the suboptimal solutions have one cost, the best lower bound VAC could produce is one. Such a small lower bound can hardly exceed the current upper bound (i.e., the best solution's cost found so far), therefore VAC performed extra consistency propagation without pruning more branches in the search than the other two.

- VAC works best when a significant lower bound can be calculated and when integer-based cost manipulations (Chapter 3) cannot produce a good enough lower bound. The QWH problem instances include many insignificant sub-optimal solutions with a cost of one, which cannot produce a good lower bound.

## 6.4 Conclusion

This chapter compared xAC's variants (i.e., TP and TP+OnOff), EDAC, and VAC on solving benchmark problems used in the literature which include UWLPs, RLFAPs, and QWHs. The empirical results showed advantages of xAC over EDAC and VAC. Section 6.1 showed that the xAC variations TP and Tp+OnOff performed one order of magnitude fewer constraint checks than EDAC. In addition, TP and TP+OnOff were comparable to each other with either performing better on some instances. Section 6.2 showed that the TP+OnOff was able to solve more problem instances within the time limit than TP, because turning off the xAC propagation when it's not needed can speed up the search process. Section 6.3 showed that the number of constraint checks performed by xAC was more than EDAC and VAC when the number of holes is small but it did not increase as fast as EDAC and VAC when the number of holes becomes larger.

We did not implement the hybrid algorithm which combines EDAC and VAC because that work is considered infeasible for this thesis (Cooper *et al.*, 2008). We also did not implement the variable orderings mentioned by Cooper et al. (2010). These variable ordering heuristics may increase the performance of EDAC and VAC. In addition, we implemented the algorithms in Java, which is slower than C and C++. However, we consider that the relative speed is more important than the absolute speed, because we can compare the relative speed of the algorithms on a common basis.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusions and Contributions

Constraint Satisfaction Problems (CSPs) and Constraint Optimization Problems (COPs) include many real-world applications in machine vision, belief maintenance, scheduling, and others. Because of the various applications in which CSPs and COPs are useful, extensive research has been devoted into developing more efficient algorithms for solving them. Soft AC propagation has been shown to be highly effective in solving COPs. Various soft AC algorithms have been designed to solve hard real-world COPs. It is important to empirically compare these algorithms on solving different problems.

The soft AC algorithms W-AC*2001, EDAC, and VAC originate from the AC3 algorithm which was designed to solve CSPs. By generalizing the classical arc consistency for solving CSPs into soft arc consistency for solving COPs, W-AC*2001 shifts costs from non-binary constraints down to unary constraints, whose costs are projected down to the zero-arity constraint. This zero-arity constraint is used as a lower bound in the branch-and-bound search. If the upper bound (i.e., the cost of the best solution found so far) is smaller than the lower bound (i.e., the least amount of cost to be paid by the best full assignments below the current partial solution), the subtree below the current partial solution is pruned from the search space. Otherwise, the branch-and-bound search keeps trying values from the subtree. By shifting costs to maintain more restrictive and stronger value supports, EDAC is able to calculate a higher lower bound than W-AC*2001, thus allowing the branch-and-bound search to prune more subtrees. By allowing rational-number-based operations, VAC is able to further increase the lower bound found by EDAC. Unlike the aforementioned three,

xAC gathers information about the costs of values from the whole constraint graph and calculates a value ordering heuristic for the current variable in the branch-and-bound search. Although xAC does not remove any value from the variables, it can calculate a good value ordering heuristic which allows the branch-and-bound search to find a good solution early during the search. Such a solution's cost is used as a new upper bound in searching the remaining constraint problem. If the upper bound is good enough, a significant proportion of the search space can be pruned.

In this work, we adapted the ACS template algorithm (Likitvivatanavong *et al.*, 2007) into a branch-and-bound search template for solving COPs. Using this branch-and-bound search template, we implemented several constraint solvers using these algorithms: W-AC*2001, EDAC, VAC, and xAC. We also implemented five variants of xAC: TreePruning, OnOff, RadicalPruning, TreePruning+OnOff, and xAC*. We compared the performance of the aforementioned algorithms on solving random and real-world problems. Chapter 4 showed that, on small problems with a range of numbers of variables and constraints, xAC produces value orderings more similar to the exact value ordering than others. Chapter 5 showed that xAC performs fewer constraint checks and searches fewer nodes than others on solving random MaxCSPs. On solving random UICOPs, xAC performs more constraint checks and searches more nodes on small problems than EDAC and VAC but fewer constraint checks and fewer nodes than VAC on larger problems. Chapter 6 showed that xAC is able to solve large and hard problems including UWLPs, RLFAPS, and Quasigroups in real world.

## 7.2 Future Work

The original version and variants of xAC provide several possible directions for performance improvements. One promising possibility is directional xAC. In VAC, a cost $\lambda$ flows along a R-path. This cost flow will remove all values of at least one domain in $Bool(P)$, which may indicate a possible direction for xAC's propagation to produce better value ordering. In some problems, we are able to calculate the same lower bound from VAC by sending a modified marginal of certain variables along the R-path. Therefore we believe that using a R-path to guide the direction of

xAC's propagation may lead to better value orderings while reducing the amount of work because it's propagating information along an informed path instead of over the whole constraint graph.

xAC produces a value ordering heuristic while EDAC removes arc inconsistent values, both of which could reduce the search space. By performing EDAC first to reduce the number of values and xAC later to produce value ordering on the remaining values, we may be able to create a hybrid which performs better than either xAC or EDAC alone.

In order to compare independent soft AC algorithms, we did not implement EDAC+VAC (Cooper *et al.*, 2010). However, since EDAC can remove values that would not be deleted by VAC, incorporating EDAC into VAC can reduce the amount of work performed by VAC while still preserving the ability to find a good lower bound. Including EDAC+VAC in the empirical experiments will provide more comprehensive results.

We use the lexicographical variable ordering in our branch-and-bound search. Although it is simple to implement, a more sophisticated variable ordering heuristic can significantly improve the speed of search (Cooper *et al.*, 2010).

# References


O. Angelsmark. Partitioning based algorithms for some colouring problems. In *In Recent Advances in Constraints, volume 3978 of LNAI*, pages 44–58. Springer Verlag, 2005.

F. Bacchus and P. Beek. On the conversion between non-binary and binary constraint satisfaction problems. *National Conference on Artificial Intelligence*, pages 311–318, Sep 1998.

R. Barták. On generators of random quasigroup problems. In *Recent Advances in Constraints*, volume 3978 of *Lecture Notes in Computer Science*, pages 164–178. 2006.

S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based csps and valued csps: Frameworks, properties, and comparison. *Constraints*, 4:199–240, Sep 1999.

J. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, pages 651–655, 1975.

A. Borning, M. Mahert, and A. Martindale. Constraint hierarchies and logic programming. In *International Conference on Logic Programming*, pages 149–164, 1989.

D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

B. Cabon, S. D. Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4:79–89, 1999.

D. Clark, J. Frank, I. Gent, E. Macintyre, N. Tomov, and T. Walsh. Local search and the number of solutions. In *Principles and Practice of Constraint Programming*, pages 119–133, 1996.

D. Cohen, M. Cooper, P. Jeavons, and A. Krokhin. A maximal tractable class of soft constraints. *Journal Of Artificial Intelligence Research*, 22:1–22, 2004.

D. A. Cohen, M. C. Cooper, and P. G. Jeavons. The complexity of soft constraint satisfaction. *Artificial Intelligence*, 170:983–1016, 2006.

M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199–227, Feb 2004.

M. Cooper, S. Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted csp. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 253–258, 2008.

M. C. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, pages 449–478, 2010.

R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. In *Constraints*, volume 6, pages 345–372, 2001.

S. D. Givry and M. Zytnicki. Existential arc consistency: Getting closer to full arc consistency in weighted csps. In *In Proceedings of the 19th IJCAI*, pages 84–89, 2005.

C. P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 221–227, 1997.

C. Gomes and D. B. Shmoys. The promise of lp to boost csp techniques for combinatorial problems. In *CP-AI-OR'02*, pages 291–305, 2002.

F. Heras, J. Larrosa, S. Givry, and M. Zytnicki. Existential arc consistency: Getting closer to full arc consistency in weighted csps. In *Proceedings of IJCAI'05*, pages 84–89, 2005.

M. Horsch and W. Havens. Probabilistic arc consistency: A connection between constraint reasoning and probabilistic reasoning. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 282–290, 2000.

M. C. Horsch, W. S. Havens, and A. K. Ghose. Generalized arc consistency with application to maxcsp and scsp instances. In *Proceedings of the Fifteenth Canadian Conference on Artificial Intelligence*, pages 104–118, 2002.

M. T. Jacobson and P. Matthews. Generating uniformly distributed random latin squares. *Journal of Combinatorial Design 4*, pages 405–437, 1996.

M. Kendall. *Rank Correlation Methods*. A Charles Griffin Title, fifth edition, September 1990.

J. Kratica, V. Filipovic, V. Sesum, and D. Tosic. Solving of the uncapacitated warehouse location problem using a simple genetic algorithm. In *Proceedings of the XIV International Conference on Material Handling and Warehousing*, pages 3.33 – 3.37, 1996.

V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, Jan 1992.

A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted csp. In *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*, pages 239–244, 2003.

J. Larrosa. Node and arc consistency in weighted csp. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 48–53, 2002.

C. Likitvivatanavong, Y. Zhang, S. Shannon, J. Bowen, and E. Freuder. Arc consistency during search. In *IJCAI'07: Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 137–142, 2007.

A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, pages 225–233, 1986.

K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *In Proceedings of Uncertainty in AI*, pages 467–475, 1999.

B. A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Reasoning*. Morgan Kaufmann, Los Altos, 1988.

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

P. Prosser. Forward checking with backmarking. *Lecture Notes in Computer Science*, 923:185–204, Nov 1995.

A. Robert, editor. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, 1999.

A. Rosenfeld, R. A. Hummel, and Zucker S. W. Scene labelling by relaxation operations. *IEEE Transactions on Systems, Man, and Cybernetics*, 6:173–184, 1976.

T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 631–637, 1995.

T. Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints ?". *Proceedings of the Eighth Conference of Uncertainty in Artificial Intelligence, USA*, pages 269–275, Nov 1992.

T. Schiex. Arc consistency for soft constraints. In *CP'00*, pages 411–424, 2000.

B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *National Conference on Artificial Intelligence*, pages 440–446, 1992.

L. Shapiro and R. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:504–519, 1981.

B. M. Smith. The brláz heuristic and optimal static orderings. In *Proceedings CP'99*, pages 405–418, 1999.

G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search for solving constraint optimization problems. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 181–187, Aug 1996.

M. Vernooy and W. S. Havens. An examination of probabilistic value-ordering heuristics. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence: Advanced Topics in Artificial Intelligence*, AI '99, pages 340–352, 1999.

D. Waltz. Understanding line drawings of scenes with shadows. In Patrick Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

# Appendix A

# RLFAP Data Files

The RLFAP instances (Cabon *et al.*, 1999) used in this work can be downloaded at:

1. ftp://ftp.cs.unh.edu/pub/csp/archive/code/benchmarks/FullRLFAP.tgz
2. ftp://ftp.cert.fr/pub/lemaitre/FullRLFAP.tgz

Each instance is described by four files in the same directory.

- File "dom.txt"

  This file describes the set of domains. Each line describes one domain. The first domain is the union of all domains. This domain is used for overview, so it is excluded from the actual model.

  Each domain is described by fixed width fields, which are separated by white spaces.

  - Field 1: Domain id
  - Field 2: Domain size
  - Field $3 \cdots n$: Specific values

  For example, suppose a line in "dom.txt" is "7 2 13 45". This shows a domain whose id is 7 (or the seventh domain of the problem). This domain has 2 values, which are 13 and 45.

- File "var.txt"

  This file uses "dom.txt" to describe the variables. Each line describes a variable by fixed width fields. Field 3 and 4 are not always present.

  - Field 1: Variable id
  - Field 2: Domain id (see "dom.txt")
  - Field 3: Pre-assigned frequency (optional)
  - Field 4: Mobility cost (index of modification cost, optional)

  Variable ids may not be consecutive. The index of the mobility cost may be $0, 1, 2, 3$ or 4. 0 means that the value of the variable is already assigned and must not be modified. Index 1 to 4 means that an initial value is assigned to the variable and may be modified with a decreasing cost. Each of the index represents an actual cost specified in "cst.txt", which will be shown later.

  For example, suppose there is a line "2 10 136 1" in "var.txt". This line shows a variable whose id is 2. This variable's domain is the tenth domain defined in "dom.txt". This variable is assigned a frequency of 136. The final field's value 1 shows that the frequency can be modified with a cost.

- File "ctr.txt"

  Each line defines a binary constraint, which consists of the following fields:

  - Field 1: Id of the first variable
  - Field 2: Id of the second variable
  - Field 3: Constraint type
  - Field 4: Operator
  - Field 5: Deviation

– Field 6: Weight index

Fields 1 and 2 use the variable ids in "var.txt". Field 3 may take value D, C, F, P, or L, which is not used by the xAC solver. Field 4 is the relational operator that should be used to compare the absolute difference of the two variables' values against the one in field 5. It can be ">" or "=", corresponding to equations 6.3 and 6.4, respectively. Therefore, the constraint is:

$$|\text{Field1}-\text{Field2}|\ \text{Field4}\ \text{Field5}$$

Field 6 is optional and is used when a constraint violation is allowed. Its value is chosen from $\{0, 1, 2, 3, 4\}$. 0 means the constraint is "hard", otherwise 1 to 4 means the constraint is "soft" with a decreasing weight in the optimization criterion. The actual costs associated with the value 1 to 4 are defined in "cst.txt".

For example, suppose there is a line "13 15 $C > 233$ 4" in "ctr.txt", which shows a binary constraint between the thirteenth variable and the fifteenth variable. The absolute value of the difference between the frequencies assigned to these two variables is preferably to be larger than 233. A cost $a_4$ (defined in "cst.txt") will be caused if the constraint is violated (i.e., the difference is no larger than 233).

- File "cst.txt"

  This file defines the optimization criterion, which can be FEAS, SPAN, CARD, or MAX. When the maximum feasibility is considered, it also specifies the four interference costs (noted $c_1, \cdots, c_4$) and the four mobility costs (noted $m_1, \cdots, m_4$).

  For example, suppose "cst.txt" specifies that $c_1 = 1000$, $c_2 = 100$, $c_3 = 10$, and $c_4 = 1$, $m_1 = 500$, $m_2 = 400$, $m_3 = 250$, $m_4 = 100$. Then the interference cost associated with weight 1 is 1000, and the one with weight 2 is 100, etc. In addition, the mobility cost associated with index 1 is 500, and the one with index 2 is 400, etc.

# Definition Index

Arc Consistency, 11
Assignments, 6

Constraint, 6
Constraint Satisfaction Problem, 5
Constraint System, 20

Equivalence-Preserving Transformation, 29
Existential Arc Consistent, 39

Latin Square, 76

Node Consistency, 11

Quasigroup, 76

Radio Link Frequency Assignment Problem, 69

Semiring, 19
Solution, 7

Uncapacitated Warehouse Location Problem, 63

Valuation Structure, 17
Valued CSP, 18
Virtual Arc Consistent, 46