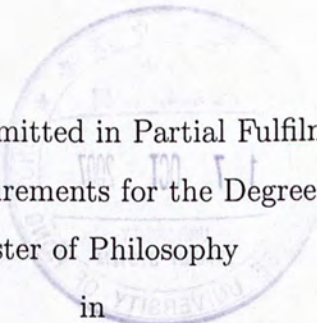


# Weighted Constraint Satisfaction with Set Variables

SIU Fai Keung



A Thesis Submitted in Partial Fulfilment  
of the Requirements for the Degree of  
Master of Philosophy  
in  
Computer Science and Engineering

©The Chinese University of Hong Kong  
August 2006

The Chinese University of Hong Kong holds the copyright of this thesis. Any person(s) intending to use a part or whole of the materials in the thesis in a proposed publication must seek copyright release from the Dean of the Graduate School.

Weighted Constant Substitution with  
Variables

Bill Lee



A Thesis Submitted to the Faculty of the Department of Mathematics  
Computer Science and Engineering

© The Chinese University of Hong Kong  
August 2007

The Chinese University of Hong Kong holds the copyright in this thesis.  
Any person(s) intending to use a part or whole of the material in this thesis  
in a proposed publication must seek prior written permission from the Dean of the  
Graduate School.

# Abstract

## Thesis/Assessment Committee

Professor Young Fung Yu (Chair)

Professor Lee Ho Man Jimmy (Thesis Supervisor)

Professor Cai Leizhen (Committee Member)

Professor Javier Larrosa (External Examiner)

# Abstract

Many problems in daily life, such as resource allocation, scheduling, timetabling, configuration and satisfiability problems, can be modeled as finite domain constraint satisfaction problems (CSPs). Set variables are ubiquitous in modeling many applications as CSPs. Various approaches to handle set variables are proposed for classical CSPs. In contrast to containing hard constraints only in classical CSPs, the ability to specify soft constraints with set variables in weighted constraint satisfaction problems (WCSPs) can enhance the expressiveness of modeling. However, efforts on practical consistency algorithms for WCSPs have only been on integer variables. The major problem associated with set variable is its high complexity. A set variable with  $n$  possible set elements has  $2^n$  set values in its domain. The time complexity to search such large domain for solutions is high. As current local consistency enforcing algorithms for WCSPs require constraints to be implemented as a cost tables, the explicit representation of set constraints in WCSPs also suffers from exponential space requirement. In this thesis, we propose compact and efficient representation schemes for set variables and common unary, binary, and ternary set constraints, as well as cardinality constraints. We adapt the classical notion of set bounds consistency for WCSPs. Instead of reasoning consistency on an entire set variable directly, we propose local consistency check at the set element level, and demonstrate that this apparent “micro”-management of consistency



does imply set bounds consistency at the variable level. In addition, we prove that our framework captures classical CSPs with set variables, and degenerates to the classical case when the weights in a problem contain only 0 and T. Last but not least, we verify the feasibility and efficiency of our proposal with a prototype implementation, the efficiency of which is competitive against ILOG Solver on classical problems and orders of magnitude better than WCSP models using 0-1 variables to simulate set variables on soft problems.

# 摘要

日常生活中的許多問題，比如資源分配、規劃、時間表安排、配置以及可滿足性問題等，都可以建模為約束滿足問題，集合變量是約束滿足問題中常見的變量類型。目前，已有許多方法處理在傳統滿足約束問題中的集合變量。相對於傳統約束滿足問題，在加權約束滿足問題中，我們可以通過加入包含集合變量的軟性約束來增強模型的表達能力。但是已有的加權約束滿足問題的相容性算法只能處理整數變量，無法處理集合變量。同時，對於一個包含  $n$  個元素的集合變量而言，其域的大小為  $2^n$ 。因此使用集合變量會增大時間和空間的複雜性。搜索答案的時間複雜性大都取決於搜索空間的大小。然而，搜索空間亦會因變量的域增大而相應增大。空間複雜性的增加是由於目前的相容性算法要求以權值分佈表來表示約束。這種明顯列出約束中權值的方法，極大地增加了空間的複雜性。針對這些問題，本文提出了高效表示集合變量、常用的一、二、三元集合約束以及集合的個數約束的方法。同時在傳統集合邊界相容性的基礎上，我們提出了在集合元素的層次上進行相容性推理的方法。我們證明了這種表面上對相容性作微觀處理的方法，可以達到集合變量上的相容性。除此之外，我們證明了我們所提出的框架，不但涵蓋了包含集合變量的傳統約束滿足問題，而且在問題中的權值只有 0 和 1 時，會退化成傳統的約束滿足問題。最後，我們用原型實驗證明了我們所提出的方法的可行性及高效性。在解決傳統問題上，我們的效率可與 ILOG Solver 相比。而在解決加權約束滿足問題上，我們的效率比以 0-1 變量來模擬集合變量的方式有更明顯的優越性。



# Acknowledgments

I would like to express my gratitude to my supervisor, Professor Jimmy Lee. He has brought me to this research area of constraint satisfaction in 2004. The path to the current thesis topic is not smooth. Several attempts were made in the past two years on other topics. Although the results of previous topics were not encouraging, I received a lot of help from Jimmy. This experience became the stepping stone for me on this thesis topic. I am very grateful for his invaluable advice and continuous support for my research.

I would like to thank my examiners Professor Javier Larrosa, Professor Fung Yu Young, and Professor Leizhen Cai. Their constructive comments help improve the quality of the thesis a lot. It was memorable for me to have oral defense with Javier while we were attending conference of AAAI-06 in Boston.

Groupmates of our research groups have contributed lots of ideas and fun for my research work. I thank Jeff Choi, Spencer Fung, Xiao Hui Ji, Gordon Lam, Yat Chiu Law, and May Woo. I enjoyed much working with them in the same office. In addition, I was often benefited by their sharing of research experience.

Last but not the least, I would like to give my best wishes to my parents and sisters for their endless support and understanding throughout my two-year master program.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	(Weighted) Constraint Satisfaction . . . . .	1
1.2	Set Variables . . . . .	2
1.3	Motivations and Goals . . . . .	3
1.4	Overview of the Thesis . . . . .	4
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Constraint Satisfaction Problems . . . . .	6
2.1.1	Backtracking Tree Search . . . . .	8
2.1.2	Consistency Notions . . . . .	10
2.2	Weighted Constraint Satisfaction Problems . . . . .	14
2.2.1	Branch and Bound Search . . . . .	17
2.2.2	Consistency Notions . . . . .	19
2.3	Classical CSPs with Set Variables . . . . .	23
2.3.1	Set Variables and Set Domains . . . . .	24
2.3.2	Set Constraints . . . . .	24
2.3.3	Searching with Set Variables . . . . .	26
2.3.4	Set Bounds Consistency . . . . .	27
<b>3</b>	<b>Weighted Constraint Satisfaction with Set Variables</b>	<b>30</b>
3.1	Set Variables . . . . .	30



3.2	Set Domains . . . . .	31
3.3	Set Constraints . . . . .	31
3.3.1	Zero-arity Constraint . . . . .	33
3.3.2	Unary Constraints . . . . .	33
3.3.3	Binary Constraints . . . . .	36
3.3.4	Ternary Constraints . . . . .	36
3.3.5	Cardinality Constraints . . . . .	37
3.4	Characteristics . . . . .	37
3.4.1	Space Complexity . . . . .	37
3.4.2	Generalization . . . . .	38
<b>4</b>	<b>Consistency Notions and Algorithms for Set Variables</b>	<b>41</b>
4.1	Consistency Notions . . . . .	41
4.1.1	Element Node Consistency . . . . .	41
4.1.2	Element Arc Consistency . . . . .	43
4.1.3	Element Hyper-arc Consistency . . . . .	43
4.1.4	Weighted Cardinality Consistency . . . . .	45
4.1.5	Weighted Set Bounds Consistency . . . . .	46
4.2	Consistency Enforcing Algorithms . . . . .	47
4.2.1	Enforcing Element Node Consistency . . . . .	48
4.2.2	Enforcing Element Arc Consistency . . . . .	51
4.2.3	Enforcing Element Hyper-arc Consistency . . . . .	52
4.2.4	Enforcing Weighted Cardinality Consistency . . . . .	54
4.2.5	Enforcing Weighted Set Bounds Consistency . . . . .	56
<b>5</b>	<b>Experiments</b>	<b>59</b>
5.1	Modeling Set Variables Using 0-1 Variables . . . . .	60
5.2	Softening the Problems . . . . .	61

5.3	Steiner Triple System . . . . .	62
5.4	Social Golfer Problem . . . . .	63
5.5	Discussions . . . . .	66
<b>6</b>	<b>Related Work</b>	<b>68</b>
6.1	Other Consistency Notions in WCSPs . . . . .	68
6.1.1	Full Directional Arc Consistency . . . . .	68
6.1.2	Existential Directional Arc Consistency . . . . .	69
6.2	Classical CSPs with Set Variables . . . . .	70
6.2.1	Bounds Reasoning . . . . .	70
6.2.2	Cardinality Reasoning . . . . .	70
<b>7</b>	<b>Concluding Remarks</b>	<b>72</b>
7.1	Contributions . . . . .	72
7.2	Future Work . . . . .	74
	<b>List of Symbols</b>	<b>76</b>
	<b>Bibliography</b>	<b>79</b>

# List of Figures

2.1	A solution to the 4-queens problem . . . . .	8
2.2	Backtracking tree search maintaining node and arc consistencies for the 4-queens problem . . . . .	15
2.3	A sample WCSP . . . . .	17
2.4	An equivalent WCSP which is NC* . . . . .	21
2.5	An equivalent WCSP which is AC* . . . . .	23
2.6	Set domain for the course selection of a student . . . . .	25
2.7	(a) An original domain for a set variable (b) A domain with 3 removed from the possible set . . . . .	27
2.8	A complete search tree for a set variable . . . . .	28
3.1	(a) $2 \in S_1 \wedge 3 \notin S_2$ (b) $S_1 = S_2$ (c) $S_1 \subseteq S_2$ . . . . .	34
3.2	An example WCSP . . . . .	35
4.1	(a) A WCSP which is not ENC (b) An equivalent WCSP which is ENC . . . . .	42
4.2	(a) A WCSP which is not EAC (b) An equivalent WCSP which is EAC . . . . .	44

# List of Tables

2.1	Projection functions for some common set constraints . . . . .	29
3.1	Space complexity of set constraints of different arities with the use of set variables and integer variables ( $e$ is the maximum number of set elements in the sets) . . . . .	38
3.2	Soft versions of common classical set constraints . . . . .	39
5.1	Runtime and number of fails for solving classical Steiner Triple System . . . . .	63
5.2	Runtime and number of fails for solving soft Steiner Triple System	63
5.3	Runtime and number of fails for the classical Social Golfer Problem	65
5.4	Runtime and number of fails for the soft Social Golfer Problem .	65



# Chapter 1

## Introduction

Many problems in daily life, such as resource allocation, scheduling, timetabling, configuration and satisfiability problems, can be modeled as finite domain constraint satisfaction problems (CSPs). When a problem is over-constrained or involves preferences, we can model the problem as a weighted constraint satisfaction problem (WCSP). In WCSP, costs are associated with tuples to reflect the quality of the assignments. On the other hand, set variables are common in modeling problems. Gervet [Ger97] demonstrated how set variables can be handled in CSPs. The idea of using set interval as a set domain and reasoning on the bounds give an efficient solving approach for CSPs with set variables.

### 1.1 (Weighted) Constraint Satisfaction

*Constraint satisfaction problems* (CSPs), defined in the sense of Mackworth [Mac77], can be briefly stated as follows :

*We are given a set of variables, a domain of possible values for each variable, and a conjunction of constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The*

*goal is to find a consistent assignment of values to the variables so that all the constraints are satisfied simultaneously.*

Constraints in *classical* CSPs can only be either fully satisfied or fully violated. In many real life applications, we have to allow partially satisfied constraints when the problems are over-constrained or involve preferences. For example, there are multiple routes taking a traveler from the origin to the destination. While the traveler can arrive at the destination via any one of the routes, the cheapest and shortest route is often the preferred choice.

The *weighted constraint satisfaction problem* (WCSP) framework, one of the soft constraint frameworks, allows us to specify preference and degree of satisfaction (or violation) by associating costs to the tuples. WCSP is thus a generalization of classical CSP. We can evaluate the quality of an assignment with the costs given by the constraints. The lower the cost is, the higher the quality of the assignment. Therefore, we are searching for the assignment with minimum cost in a WCSP.

## 1.2 Set Variables

Integer variables suffice to model many combinatorial problems. In some cases, however, unknowns in a problem can have set as values. For example, we might be interested in finding what nurses should be serving in a particular shift in a nurse rostering problem. A set variable in classical CSP takes on set values. Since the domain size of a set variable is large, Gervet [Ger97] proposes that the domain of a set variable is specified with a lower and upper bounds, which are ordered by set inclusion. Any set which falls within the bounds is in the domain of the set variable. The set constraints are composed of set relations, such as subset equal ( $\subseteq$ ) and equality ( $=$ ), and set operators, such as union ( $\cup$ ),



intersection ( $\cap$ ) and different ( $\setminus$ ). Cardinality of a set variable can be restricted with cardinality constraints. As set domain is specified as bounds, Gervet introduces an approach to reason the domains on its bounds with respect to the constraints and define set bounds consistency notions [Ger97].

### 1.3 Motivations and Goals

The need for set variables is no exception with WCSPs. Our goal is to define set variables for WCSPs as there are no existing framework for WCSPs to deal with sets. A set variable with  $n$  possible set elements has a domain of size  $2^n$ . Domain consistency techniques [Lar02, LS03] developed for integer variables cannot be practically adapted for set variables since these techniques require all elements of a variable domain to be represented explicitly. Following Gervet [Ger97], we propose efficient set bounds consistency techniques in WCSPs for set variables which reason only on the bounds of the variable domains [LS06].

Constraints in WCSPs are cost functions, mapping tuples to costs. Instead of specifying the cost functions at the tuples (of set values) level, we devise a general scheme for representing tuple costs according to costs associated with the existence and inexistence of elements in the set values. This scheme is compact and allows us to specify cost functions to all common set constraints, and degenerates to classical CSPs with set variables when all costs are either 0 or  $\top$ . Node, arc, hyper-arc, and cardinality consistency notions and the associated enforcement algorithms are defined for unary, binary, ternary, and cardinality constraints at the set element level respectively. We show that these element consistencies imply set bounds consistency [Ger97, MM97, HLS05] generalized for WCSPs. We construct a prototype implementation of our algorithms by modifying the ToolBar WCSP solver [BHdG<sup>+</sup>04]. Experiments are conducted to compare our implementation against ILOG Solver [ILO03] on classical set

CSPs, and against 0-1 variable emulation of set variables in ToolBar on softened versions of the same classical benchmarks. Results confirm that our implementation is more efficient than ILOG Solver on classical problems and two orders of magnitude better than WCSP models using 0-1 variable to simulate set variables on soft problems.

## 1.4 Overview of the Thesis

The rest of the thesis is organized as follows. Chapter 2 provides the background to the thesis. We formally introduce classical CSPs and WCSPs, and present the common solution techniques : backtracking tree search for classical CSPs and branch and bound search for WCSPs. Overview of consistency notions, including node and arc consistencies, are given for both classical and WCSPs. We also describe the use of set variables and the notion of set bounds consistency in classical CSPs. In Chapter 3, we give the formal definition of WCSPs with set variables. The representation schemes for set variables and set constraints are described. We illustrate the approach to specify costs for set constraints via cost functions at the element level. Local consistency notions for WCSPs with set variables are presented in Chapter 4. The local consistency notions include element level consistencies and weighted cardinality consistency. On top of them, we introduce weighted set bounds consistency which is implied by maintaining the above consistencies. Complexity analysis and proofs of correctness of the consistency algorithms are given. In Chapter 5, we report experimental results. We compare the performance of our prototype implementation with ToolBar [BHdG<sup>+</sup>04], a generic WCSP solver, and ILOG Solver 6.0 [ILO03], a classical CSP solver. Chapter 6 presents a review of related work on other consistency notions in WCSPs and current approaches to



handle set variables in classical CSPs. Chapter 7 concludes the thesis by summarizing our contributions and shedding light on possible directions of future research.

## Chapter 2

# Background

This chapter provides background to the thesis. We describe concepts of both classical and weighted constraint satisfaction. In particular, we introduce set variables in modeling problems for classical CSPs. We also illustrate the benefits of enforcing local consistencies in the solution searching process.

### 2.1 Constraint Satisfaction Problems

A (*classical*) *constraint satisfaction problem* (CSP) is a tuple  $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$  where  $\mathcal{X}$  is a finite set of *variables*  $\{x_1, \dots, x_n\}$ ,  $\mathcal{D}$  is a finite set of *variable domains*  $\{D(x_1), \dots, D(x_n)\}$ , and  $\mathcal{C}$  is a finite set of *constraints*. A variable  $x_i \in \mathcal{X}$  can only be assigned with a value from its variable domain  $D(x_i) \in \mathcal{D}$ . The *initial domain*  $D_0(x_i)$  for each variable  $x_i \in \mathcal{X}$  is the domain given once a CSP is defined. Each constraint  $C_{i_1, \dots, i_k} \in \mathcal{C}$  restricts the values that can be taken by the variables  $x_{i_1}, \dots, x_{i_k}$  simultaneously. In this thesis, we consider only *finite domain CSPs* in which each variable domain is *finite*. Without loss of generality, the variable domains only contain integers, though they can contain values of any types in general.

An *assignment*  $x_i \mapsto a$  assigns the value  $a \in D(x_i)$  to the variable  $x_i$ . A *tuple*  $t$  contains the assignments for a set of variables  $\{x_{i_j} \mapsto a_j \mid 1 \leq j \leq k\}$

where  $\{x_{i_1}, \dots, x_{i_k}\} \subseteq \mathcal{X}$  and  $a_j \in D(x_{i_j})$  for  $1 \leq j \leq k$ . We denote the variables in the tuple  $t$  by  $\text{var}(t)$ . A *complete assignment* is a tuple containing assignments for all variables in  $\mathcal{X}$ .

A *classical constraint*  $C_{i_1, \dots, i_k} \in \mathcal{C}$  is a function which maps  $D(x_{i_1}) \times \dots \times D(x_{i_k})$  to  $\{\text{true}, \text{false}\}$ . The set of variables  $\{x_{i_1}, \dots, x_{i_k}\}$  is a subset of  $\mathcal{X}$ . It is the *scope* of the constraint  $C_{i_1, \dots, i_k}$  and denoted by  $\text{var}(C_{i_1, \dots, i_k})$ . Without loss of generality, we denote  $C_{i_1, \dots, i_k}$  as a conjunction of all the constraints with scope  $\{x_{i_1}, \dots, x_{i_k}\}$  in a problem.

A *projection*  $t \downarrow_V$  of a tuple  $t$  to a set of variables  $V \subseteq \text{var}(t)$  is a tuple  $t'$  such that  $t' \subseteq t$  and  $t'$  involves only variables in  $V$ . We abuse the notation of constraint  $C_{i_1, \dots, i_k}$  to take also a tuple  $t = \{x_{i_1} \mapsto a_1, \dots, x_{i_k} \mapsto a_k\}$  as an argument such that  $C_{i_1, \dots, i_k}(t) = C_{i_1, \dots, i_k}(a_1, \dots, a_k)$ . Given a constraint  $C$  and a tuple  $t$ , where  $\text{var}(C) \subseteq \text{var}(t)$ , the tuple  $t$  *satisfies*, or *consistent* with, the constraint  $C$  if and only if  $C(t \downarrow_{\text{var}(C)}) = \text{true}$ . Conversely, the tuple  $t$  *violates*, or *inconsistent* with, the constraint  $C$  if and only if  $C(t \downarrow_{\text{var}(C)}) = \text{false}$ .

A *solution* to a CSP is a complete assignment which satisfies all the constraints in  $\mathcal{C}$  simultaneously. In solving a CSP, we are searching for solutions to the problem.

### Example 2.1 The $n$ -queens problem

The  $n$ -queens problem is to place  $n$  queens on a  $n \times n$  chessboard so that they do not attack one another. We can model the problem as a CSP by using  $n$  variables  $x_1, \dots, x_n$  for each column of the chessboard. The value for variable  $x_i$  denotes the position, in terms of row number, of the  $i$ -th queen in the  $i$ -th column of the chessboard. Thus, each variable domain is  $\{1, \dots, n\}$ . For each pair of columns  $(i, j)$  on the board, where  $i, j \in \{1, \dots, n\}$  and  $i \neq j$ , we have two constraints as follows :

1.  $x_i \neq x_j$



$$2. |x_i - x_j| \neq |i - j|$$

The first constraint forbids pair of queens to be located in the same row. The model inherently does not allow any pair of queens to be located in the same column as each variable can only take one value. The second constraint forbids pair of queens to be placed in the same diagonal on the chessboard. One of the solutions for 4-queens problem is depicted in Figure 2.1. The  $4 \times 4$  squares represent a chessboard. Each letter Q represents a queen. The variable for each column is labeled above the corresponding column while the values are marked in the left hand side of the chessboard according to the row number. Thus, the figure shows the solution  $\{x_1 \mapsto 2, x_2 \mapsto 4, x_3 \mapsto 1, x_4 \mapsto 3\}$ .

	$x_1$	$x_2$	$x_3$	$x_4$
1			Q	
2	Q			
3				Q
4		Q		

Figure 2.1: A solution to the 4-queens problem

■

### 2.1.1 Backtracking Tree Search

A CSP can be solved by systematic search. The solution space of the problem is traversed systematically as a tree structure. This method guarantees to find all the solutions if the problem has ones. Otherwise it proves unsatisfiability of the problem. Thus, systematic search is both sound and complete. In practice, *backtracking tree search* algorithm [GB65, Gas77, BP81, DP87, Nad89], one of systemic search algorithms, is used to solve CSPs. The backtracking tree



search described below traverses the tree of possible assignments in a *depth-first* manner. Algorithm 2.1 gives the procedure for backtracking tree search given in [Apt03] for finding a single solution.

---

**Algorithm 2.1:** Backtracking tree search

---

```

1 Procedure backtrack( $t, j, \mathcal{D}, \text{success}$ )
2 begin
3   while  $D(x_j) \neq \emptyset$  and  $\neg \text{success}$  do
4      $a \in D(x_j)$ 
5      $D(x_j) := D(x_j) \setminus \{a\}$ 
6     if  $\text{cons}(t, x_j \mapsto a)$  then
7        $t := t \cup \{x_j \mapsto a\}$ 
8        $\text{success} := (j = n)$ 
9       if  $\neg \text{success}$  then
10        backtrack( $t, j + 1, \mathcal{D}, \text{success}$ )
11 end

12 begin
13    $\text{success} := \text{false}$ 
14    $t := \emptyset$ 
15   backtrack( $t, 1, \mathcal{D}, \text{success}$ )
16 end

```

---

The search starts with an empty tuple of assignment  $t$ . In the algorithm, it incrementally extends the tuple with assignments. The order of choosing variables and values during search can be arbitrary, but experiments and analysis shows that applying ordering heuristics can affect the efficiency of the search in many cases [BR75, Pur83, SS87, HE80, ZM88].

While there are unassigned variables, also known as *future variables*, the search chooses a variable with a value from the corresponding variable domain. The selected value is removed from the domain to avoid choosing the same value again. The function  $\text{cons}()$  checks if the current selected pair of variable and value is consistent with the assignments in the tuple with respect to the

constraints. If it is consistent, the new assignment is committed. Otherwise another value is selected for the same process. The procedure proceeds with the next unassigned variable, if any. When all the variables are assigned with values, a solution is found.

In case of there is no alternative value in the domain, the search backtracks to the previous state. The assignment of previous variable is undone and the search considers other values in the domain for the previous variable. If the search backtracks to the first variable with empty domain, then there is no solution to the problem.

The search stops once it finds the first solution of the problem, but it can be modified easily to search for all solutions.

### 2.1.2 Consistency Notions

A standard backtracking tree search, which described in previous subsection, has some major drawbacks [Bar99]. One of the drawbacks is late detection of the conflict among the assignments. Many studies are done to detect the *inconsistency* sooner.

A CSP is a tuple  $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$  with a set of variables  $\mathcal{X}$ , a set of variable domains  $\mathcal{D}$  and a set of constraints  $\mathcal{C}$ . In the following, we introduce two common consistency notions : *node consistency* and *arc consistency*.

#### Node Consistency

Node consistency [Mac77] deals with unary constraints. A variable  $x_i$  is *node consistent* if and only if  $\forall a \in D(x_i), C_i(a)$  is satisfied. A CSP is *node consistent* if all variables are node consistent.

Algorithm 2.2 shows a procedure to enforce node consistency [Mac77]. For each variable  $x_i$ , it retains only those domain values in  $D(x_i)$  which satisfy the

unary constraint  $C_i$ .

---

**Algorithm 2.2:** The node consistency algorithm

---

```

1 Procedure NC( $i$ )
2 begin
3   |  $D(x_i) := D(x_i) \cap \{a \mid C_i(a)\}$ 
4 end

5 begin
6   | for  $i := 1$  to  $n$  do
7     |   | NC ( $i$ )
8 end

```

---

**Example 2.2** Given variable  $x$ , its domain  $D(x) = \{1, 2, 3, 4, 5\}$ , and a unary constraint  $x < 4$ . The variable  $x$  is not node consistent as  $x \mapsto 4$  and  $x \mapsto 5$  do not satisfy the constraint. If the values 4 and 5 are removed from the domain  $D(x)$ , now with  $D(x) = \{1, 2, 3\}$ , the variable  $x$  becomes node consistent. ■

### Arc Consistency

Arc consistency [Mac77] deals with binary constraints. A pair of variables  $(x_i, x_j)$ , where  $i \neq j$ , is *arc consistent* if and only if  $\forall a \in D(x_i)$ , such that  $C_i(a)$  is satisfied, there is a value  $b \in D(x_j)$  such that  $C_j(b)$  and  $C_{i,j}(a, b)$  are satisfied. The value  $b \in D(x_j)$  is the *binary support* of the value  $a \in D(x_i)$  with respect to  $C_{i,j}$ . A CSP is *arc consistent* if pair of variables are arc consistent.

A basic arc consistency enforcing algorithm, AC-1, is depicted in Algorithm 2.3 [Mac77]. It first maintains node consistency for all variables. A queue is then initialized with all the variable pairs which have corresponding binary constraints in the CSP. For each pair of variables  $(x_i, x_j)$ , the function `Revise()` removes any value in  $D(x_i)$  which does not have binary support in  $D(x_j)$  with respect to  $C_{i,j}$ . However, later when the algorithm revises  $D(x_j)$



---

**Algorithm 2.3:** The first arc consistency algorithm (AC-1)
 

---

```

1 Procedure Revise( $i, j$ )
2 begin
3   delete := false
4   for  $a \in D(x_i)$  do
5     if  $\nexists b \in D(x_j)$  such that  $C_{i,j}(a, b) = \text{true}$  then
6        $D(x_i) := D(x_i) \setminus \{a\}$ 
7       delete := true
8   return delete
9 end

10 Procedure AC-1()
11 begin
12   for  $i := 1$  to  $n$  do
13     NC( $i$ )
14    $Q := \{(i, j) \mid C_{i,j} \in \mathcal{C}\}$ 
15   repeat
16     change := false
17     for  $(i, j) \in Q$  do
18       change := (Revise( $i, j$ ) or change)
19   until  $\neg$ change
20 end

```

---

for the variable pair  $(j, k)$ , some values in  $D(x_j)$  maybe removed for arc consistency. These removed values from  $D(x_j)$  may be the original binary supports for values in  $D(x_i)$ . As the values in  $D(x_i)$  may lose their supports, the algorithm iteratively checks all pairs of variables in the queue until there is no change in a single pass to ensure all the values in the domains have corresponding binary supports.

---

**Algorithm 2.4:** The third arc consistency algorithm (AC-3)

---

```

1 Procedure AC-3()
2 begin
3   for  $i := 1$  to  $n$  do
4      $\lfloor$  NC( $i$ )
5      $Q := \{(i, j) \mid C_{i,j} \in \mathcal{C}\}$ 
6     while  $Q \neq \emptyset$  do
7        $(k, m) \in Q$ 
8        $Q := Q \setminus \{(k, m)\}$ 
9       if Revise( $k, m$ ) then
10         $\lfloor Q := Q \cup \{(i, k) \mid C_{i,k} \in \mathcal{C} \wedge i \neq m\}$ 
11 end

```

---

The algorithm AC-1 is inefficient because a single change in a variable domain leads to an additional pass in the algorithm. In many cases, only a small subset of variable domains are affected by each single change. A more efficient arc consistency enforcing algorithm, AC-3, is designed to revise only the affected variable domains in case of any changes. Algorithm 2.4 shows the AC-3 algorithm [Mac77]. It is similar to AC-1 except it removes a variable pair from the queue each time before checking for supports. When there is a change in a variable domain  $D(x_k)$ , it inserts back only the variable pairs which contain variable  $x_k$  in the scope of the corresponding binary constraint to the queue.

There are more sophisticated and efficient arc consistency enforcing algorithms which include AC-4 [MH86], AC-5 [Per92], AC-6 [Bes94], AC-7 [BFR99], and AC2001 [BR01, ZY01, BRYZ05], but the fundamental concepts are the similar. They all remove values from variable domains in a CSP to maintain arc consistency.

**Example 2.3** Given a CSP with variables  $x_1$  and  $x_2$ , and domains  $D(x_1) = \{1, 2, 3, 4, 5\}$  and  $D(x_2) = \{1, 2, 3\}$ . We consider the constraint  $x_1 - x_2 = 2$ . Variable pair  $(x_1, x_2)$  is not arc consistent while variable pair  $(x_2, x_1)$  is. It is because there are no values in  $D(x_2)$  which satisfies the constraint for  $x_1 \mapsto 1$  and  $x_1 \mapsto 2$ . By removing 1 and 2 from  $D(x_1)$ , the variable pair  $(x_1, x_2)$  becomes arc consistent with the constraint. ■

Figure 2.2 shows a complete backtracking search tree for the 4-queens problem when node and arc consistencies are enforced. The search chooses variables and values in lexicographic order. The node and arc consistencies enforcing algorithms are incorporated to the backtracking tree search algorithm such that consistency check and domain pruning are carried out after each variable assignment. In the figure, each recent assignment is labeled on the edge connecting the previous node and the current node. The letter X denotes a value being removed from the domain due to inconsistency. A leaf node marked as fail when the node has empty domain. Otherwise the leaf node represents a solution to the problem.

## 2.2 Weighted Constraint Satisfaction Problems

A *weighted constraint satisfaction problem* (WCSP) is a specific subclass of *valued CSP* [SFV95] which associates costs to tuples. The costs are specified by a *valuation structure*. The preferences in the problem can be expressed in terms



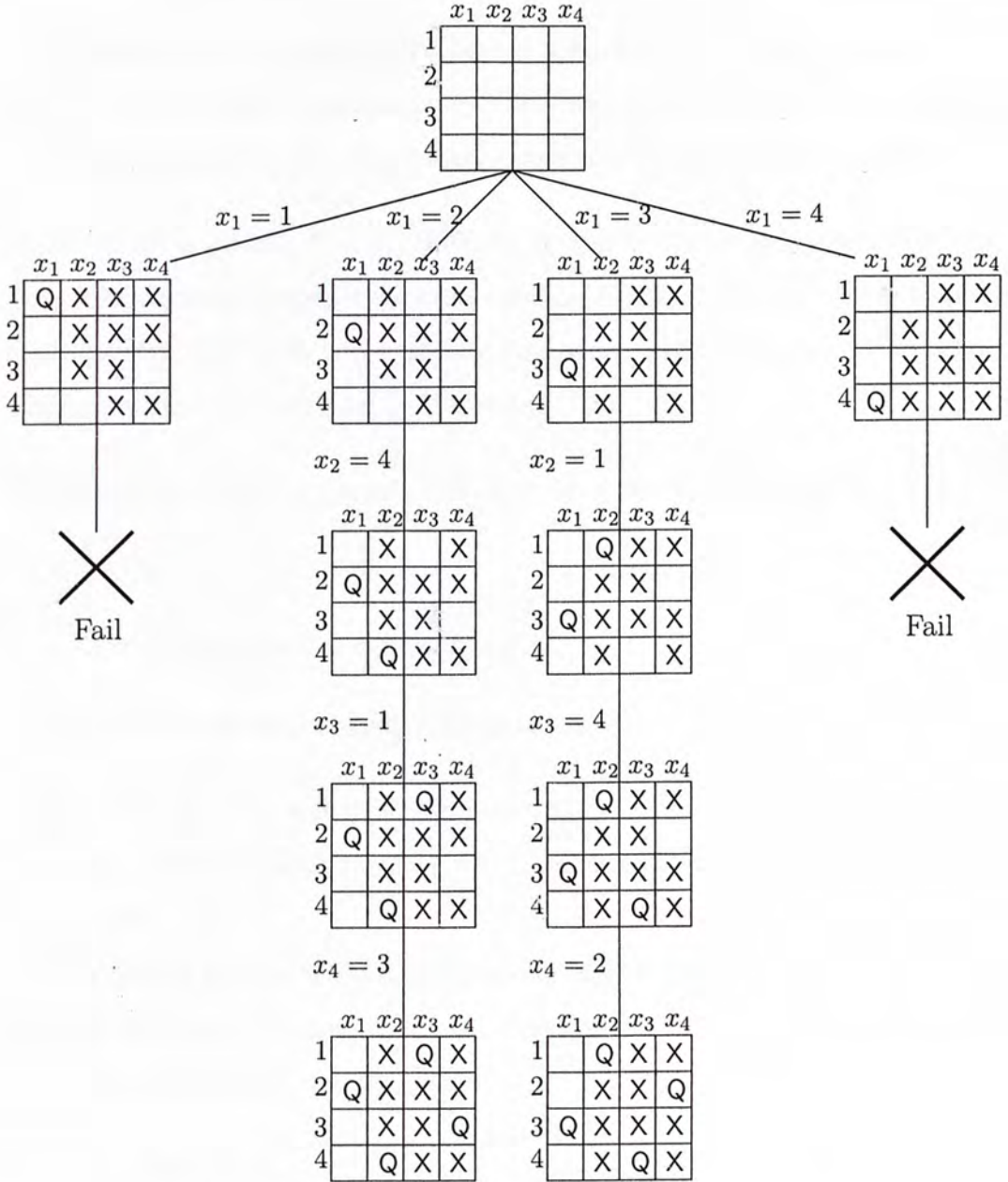


Figure 2.2: Backtracking tree search maintaining node and arc consistencies for the 4-queens problem

of costs. Thus, the WCSP framework provides a way to model optimization problem.

**Definition 2.1** A *valuation structure* is a triple  $S = (E, \oplus, \succeq)$ , where  $E$  is the set of costs totally ordered by  $\succeq$ . The maximum and the minimum costs are  $\top$  and  $\perp$  respectively. The binary operation  $\oplus$  on  $E$  combines costs.

A WCSP is a tuple  $\mathcal{P} = (k, \mathcal{X}, \mathcal{D}, \mathcal{C})$ .  $\mathcal{X}$  and  $\mathcal{D}$  are the set of variables and the set of domains respectively as in classical CSPs.  $\mathcal{C}$  is a set of constraints. Each constraint  $C \in \mathcal{C}$  is a cost function which maps assignments to costs. The valuation structure used in WCSPs is  $S(k)$ .

**Definition 2.2**  $S(k) = ([0, 1, \dots, k], \oplus, \geq)$  is a valuation structure, where

- $k \in [1, \dots, \infty]$
- $\oplus$  is defined as  $a \oplus b = \min\{k, a + b\}$
- $\geq$  is the standard order among naturals

In a WCSP with valuation structure  $S(k)$ , we have  $\perp = 0$  and  $\top = k$ . There is a zero-arity constraint  $C_\emptyset$  which represents the global lower bound of the WCSP.

The cost of a tuple  $t$ ,  $\mathcal{V}(t)$ , is a measure of quality of the tuple. The lower the cost, the higher the quality. It is defined as the sum of all costs associated with the constraints in the problem,

$$\mathcal{V}(t) = \sum_{C_{i_1, \dots, i_n} \in \mathcal{C}, \{x_{i_1}, \dots, x_{i_n}\} \subseteq \text{var}(t)} C_{i_1, \dots, i_n}(t \downarrow_{x_{i_1}, \dots, x_{i_n}}) \oplus C_\emptyset$$

The formula above is a slightly generalized form of Larrosa's definition [Lar02], which restricts discussion on only binary WCSPs. In this thesis, we do not restrict the arity of the constraints.

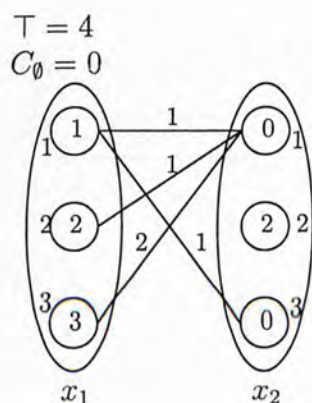


Figure 2.3: A sample WCSP

A tuple  $t$  is *consistent* if  $\mathcal{V}(t) < \top$ . In solving WCSPs, we are searching for *solutions* which are complete consistent assignments with minimum cost.

Figure 2.3 shows a sample WCSP. In the figure, each oval represents a variable. The circles inside an oval are the domain values which are labeled besides the circles. The integers in the circles are the unary costs to the corresponding values for the variable. A line joining two values in two variable domains represents a binary constraint with costs given above the line.

### 2.2.1 Branch and Bound Search

WCSPs are usually solved by *branch and bound* search, which is a solving technique for optimization problems. The search procedure is similar to tree search in solving CSPs. However, it keeps the cost of a complete solution found so far. Initially, the global lower bound,  $C_\emptyset$ , is set to 0 and the global upper bound,  $\top$ , is set to  $\infty$ . After each variable assignment, the search evaluates the current lower bound. If the current lower bound is higher than or equal to  $\top$ , the search backtracks to the previous assignment. Otherwise, the search proceeds with another variable assignment. Once a complete assignment with



cost less than  $\top$  is found,  $\top$  is set to the cost of the assignment. Therefore, the search keeps narrowing the search space to find an optimal solution.

---

**Algorithm 2.5: Branch and bound search**


---

```

1 Procedure LookAhead( $i \mapsto a, \mathcal{C}'$ )
2 begin
3    $\mathcal{C}' := \mathcal{C}' \setminus \{C_i\}$ 
4   for  $C_{i,j} \in \mathcal{C}'$  do
5     for  $b \in D(x_j)$  do
6        $C_j(b) := C_j(b) \oplus C_{i,j}(a, b)$ 
7      $\mathcal{C}' := \mathcal{C}' \setminus \{C_{i,j}\}$ 
8 end

9 Procedure BranchAndBound( $t, v_t, k, \mathcal{X}, \mathcal{D}, \mathcal{C}$ )
10 begin
11   if  $X = \emptyset$  then
12     return  $C_\emptyset$ 
13   else
14      $x_i \in X$ 
15     for  $a \in D(x_i)$  do
16        $\mathcal{D}' := \mathcal{D}$ 
17        $\mathcal{C}' := \mathcal{C}$ 
18        $t' := t \cup \{x_i \mapsto a\}$ 
19        $v_{t'} := v_t \oplus C_i(a)$ 
20       LookAhead( $i \mapsto a, \mathcal{C}'$ )
21       if LocalConsist( $k, \mathcal{X} \setminus \{x_i\}, \mathcal{D}', \mathcal{C}'$ ) then
22          $k := \text{BranchAndBound}(t', v_{t'}, \mathcal{X} \setminus \{x_i\}, \mathcal{D}', \mathcal{C}')$ 
23     return  $k$ 
24 end

```

---

Algorithm 2.5 shows a branch and bound search procedure [LS04]. The tuple  $t$  contains variable assignments. The cost of the tuple is  $v_t$ . After selected a variable and a value for the current assignment, `LookAhead()` transforms the current problem to a subproblem in which the variable  $x_i$  is assigned with  $a$ . `LocalConsist()` checks for local consistency for the transformed problem.

The search proceeds to another variable assignment only when the problem is consistent.

## 2.2.2 Consistency Notions

Similar to the classical case, maintaining local consistencies can also reduce the search space in WCSPs. Common local consistencies in WCSPs are *star node consistency* and *star arc consistency*. More sophisticated consistency notions include directional arc consistency [LS03], full directional arc consistency [LS03], and existential arc consistency [dGHZL05].

A WCSP is a tuple  $\mathcal{P} = (k, \mathcal{X}, \mathcal{D}, \mathcal{C})$  which associated with the valuation structure  $S(k)$ .  $\mathcal{X}$  and  $\mathcal{D}$  are a set of variables and a set of variable domains.  $\mathcal{C}$  is a set of constraints. Two WCSPs are said to be *equivalent* to each other if they contain the same set of variables and define the same cost distribution on complete assignments [dGHZL05]. A consistency enforcing algorithm transforms a WCSP  $P$  to an equivalent WCSP  $P'$  such that  $P'$  satisfies the requirement of the corresponding consistency notion. Consistencies are enforced by applying pruning inconsistent values and forcing supports. Supports can be forced by *sending* costs between the constraints [CS04]. Subtraction is a useful operation on costs when forcing supports.

**Definition 2.3** Let  $a, b \in \{0, \dots, k\}$  such that  $a \geq b$ . The *subtraction* of  $b$  from  $a$  [Lar02, LS04] is defined as :

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

### Node Consistency

A value  $a \in D(x_i)$  of variable  $x_i$  is *star node consistent* ( $NC^*$ ) [LS03] with respect to  $C_i$  if  $C_\emptyset \oplus C_i(a) < \top$ . Variable  $x_i$  is  $NC^*$  with respect to  $C_i$  if :

- all its values are  $NC^*$ , and
- $\exists a \in D(x_i)$  such that  $C_i(a) = \perp$ .

Value  $a$  is a *unary support* for the variable  $x_i$ . The WCSP is  $NC^*$  if every variable is  $NC^*$ .

---

**Algorithm 2.6:**  $NC^*$  algorithm
 

---

```

1 Procedure  $NC^*(\mathcal{X}, \mathcal{D}, C)$ 
2 begin
3   for  $x_i \in \mathcal{X}$  do
4      $v := \operatorname{argmin}_{a \in D(x_i)} \{C_i(a)\}$ 
5      $\alpha := C_i(v)$ 
6      $C_\emptyset := C_\emptyset \oplus \alpha$ 
7     for  $a \in D(x_i)$  do
8        $C_i(a) := C_i(a) \ominus \alpha$ 
9   for  $x_i \in \mathcal{X}$  do
10    for  $a \in D(x_i)$  do
11      if  $C_i(a) \oplus C_\emptyset = \top$  then
12         $D(x_i) := D(x_i) \setminus \{a\}$ 
13 end
  
```

---

Algorithm 2.6 describes an algorithm for enforcing  $NC^*$  [Lar02]. For each variable  $x_i$  in the problem, the algorithm finds the value  $v$  with minimum unary cost by  $\operatorname{argmin}$ . The unary cost  $C_i(v)$  is added to the global lower bound  $C_\emptyset$  as it is the minimum cost for the unary constraint  $C_i$ . This cost is subtracted from all the unary costs in  $C_i$  to maintain equivalence. A consistency check is performed to remove any value with total cost,  $C_i \oplus C_\emptyset$ , equals to  $\top$ .

**Example 2.4** The variable  $x_1$  in WCSP in Figure 2.3 is not node consistent. The equivalent WCSP, which is node consistent, is shown in Figure 2.4. Cost 1 is subtracted from the unary cost of each value in  $D(x_1)$  and sent to  $C_\emptyset$ . ■



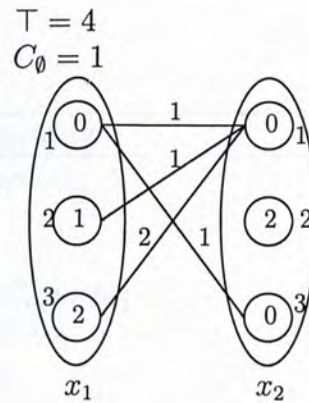


Figure 2.4: An equivalent WCSP which is NC\*

### Arc Consistency

A value  $a \in D(x_i)$  is *arc consistency (AC)* [LS03] with respect to constraint  $C_{ij}$  if it is node consistent and  $\exists b \in D(x_j)$  such that  $C_{i,j} = \perp$ .

Value  $b$  is a *binary support* of the value  $a$ . Variable  $x_i$  is AC if all its values are AC with respect to constraint  $C_{i,j}$ . The WCSP is AC\* if every variable is AC and NC\*.

Algorithm 2.7 shows the pseudocode for maintaining AC\* [LS04]. The algorithm holds a list of variables of the problem. For each variable  $x_i$ , the algorithm finds a support for each value  $a$  in the variable domain  $D(x_i)$  with respect to constraint  $C_{i,j}$  in `FindSupports()`. The support  $b$  is found so that the minimum cost for  $C_{i,j}(a, b)$  with  $x_i \mapsto a$  is  $\perp$ . The minimum cost is added to the unary cost  $C_i(a)$ . This cost also subtracted from all the binary costs  $C_{i,j}(a, c)$  for all values  $c \in D(x_j)$ . Lastly, the algorithm removes any inconsistent value  $a \in D(x_i)$  such that  $C_i(a) = \top$ .

**Example 2.5** The WCSP in Figure 2.4 is NC\* but not AC\*. Since for every value  $a \in D(x_1)$ , the cost for  $C_{1,2}(a, 1)$  is larger than  $\perp$ . The minimum binary cost, which is 1, is subtracted from the binary constraint and sent to the unary

---

**Algorithm 2.7: AC\* algorithm**


---

```

1 Procedure FindSupports( $i, j$ )
2 begin
3   for  $a \in D(x_i)$  do
4      $\alpha := \min_{b \in D(x_j)} \{C_{i,j}(a, b)\}$ 
5      $C_i(a) := C_i(a) \oplus \alpha$ 
6     for  $b \in D(x_j)$  do
7        $C_{i,j}(a, b) := C_{i,j}(a, b) \ominus \alpha$ 
8 end

9 Procedure PruneVar( $i$ )
10 begin
11   change := false
12   for  $a \in D(x_i)$  do
13     if  $C_i(a) = \top$  then
14        $D(x_i) := D(x_i) \setminus \{a\}$ 
15       change := true
16   return change
17 end

18 Procedure AC*( $\mathcal{X}, \mathcal{D}, \mathcal{C}$ )
19 begin
20    $Q := \{1, 2, \dots, n\}$ 
21   while  $Q \neq \emptyset$  do
22      $j \in Q$ 
23      $Q := Q \setminus \{j\}$ 
24     for  $C_{i,j} \in \mathcal{C}$  do
25       FindSupports( $i, j$ )
26       if PruneVar( $i$ ) then
27          $Q := Q \cup \{i\}$ 
28 end

```

---

cost of  $x_2 \mapsto 1$ . Figure 2.5 is the equivalent WCSP which is NC\* and AC\*.

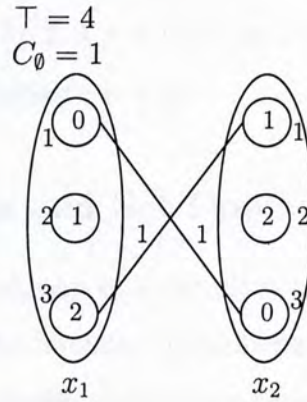


Figure 2.5: An equivalent WCSP which is AC\*

■

## 2.3 Classical CSPs with Set Variables

A set is a collection of distinct objects, and is characterized by what elements belong to it and what elements do not. Each set is associated with a cardinality which is the number of elements in the set. For example, the set  $S = \{1, 2, 3, 5\}$  has a cardinality  $|S|$  equal to four. In particular, the integer 1 belongs to the set while the integer 4 does not. In this thesis, we restrict our discussion on finite integer set variables in which all set domains contains values of finite sets.

Many problems can be naturally modeled with set variables. Suppose we are modeling the courses taken by a student in a semester. The student needs to take two compulsory courses and two elective courses. The six available courses are represented by integers from 1 to 6. Suppose the compulsory courses are denoted by the integers 1 and 3 while 2, 4, 5 and 6 are the



numbers denoting the elective courses. We can use a set variable  $S$  to represent the courses taken by the student. The corresponding set domain is  $D(S) = \{u \mid \{1, 3\} \subseteq u \subseteq \{1, 2, 3, 4, 5, 6\}\}$ . In addition, we also requires that the cardinality of the set variable is four.

### 2.3.1 Set Variables and Set Domains

A set variable which can take up to  $n$  set elements has domain size  $2^n$ . If we model a set variable with the domain containing all the possible set values, the time and space complexity makes solution searching impractical. In practice, a set variable  $S$  has a set domain represented as an interval, which is bounded by a *required set*  $RS(S)$  and a *possible set*  $PS(S)$ . The required set and possible set are also known as *greatest lower bound* and *least upper bound* of the set domain respectively. The required set contains elements which must exist in the set. In contrast, the possible set contains any elements which may exist in the set. Any element does not in the possible set must not exist in the set. It is clear that  $RS(S) \subseteq PS(S)$  and any set  $u$  such that  $RS(S) \subseteq u \subseteq PS(S)$  is in the set domain. We denote a set domain bounded by  $RS(S)$  and  $PS(S)$  as  $[RS(S), PS(S)]$ .

In the previous example, the required set is  $\{1, 3\}$  and the possible set is  $\{1, 2, 3, 4, 5, 6\}$ . The domain bounded by these two sets is shown in Figure 2.6. Each arc in the figure represents a subset relation such that the set value below is a subset of the set value above on two end points of the arc.

### 2.3.2 Set Constraints

Set constraints are composed of common set relations and set operators. Set relations include subset ( $\subseteq$ ) and equality ( $=$ ). Set operators include union ( $\cup$ ), intersection ( $\cap$ ), difference ( $\setminus$ ).

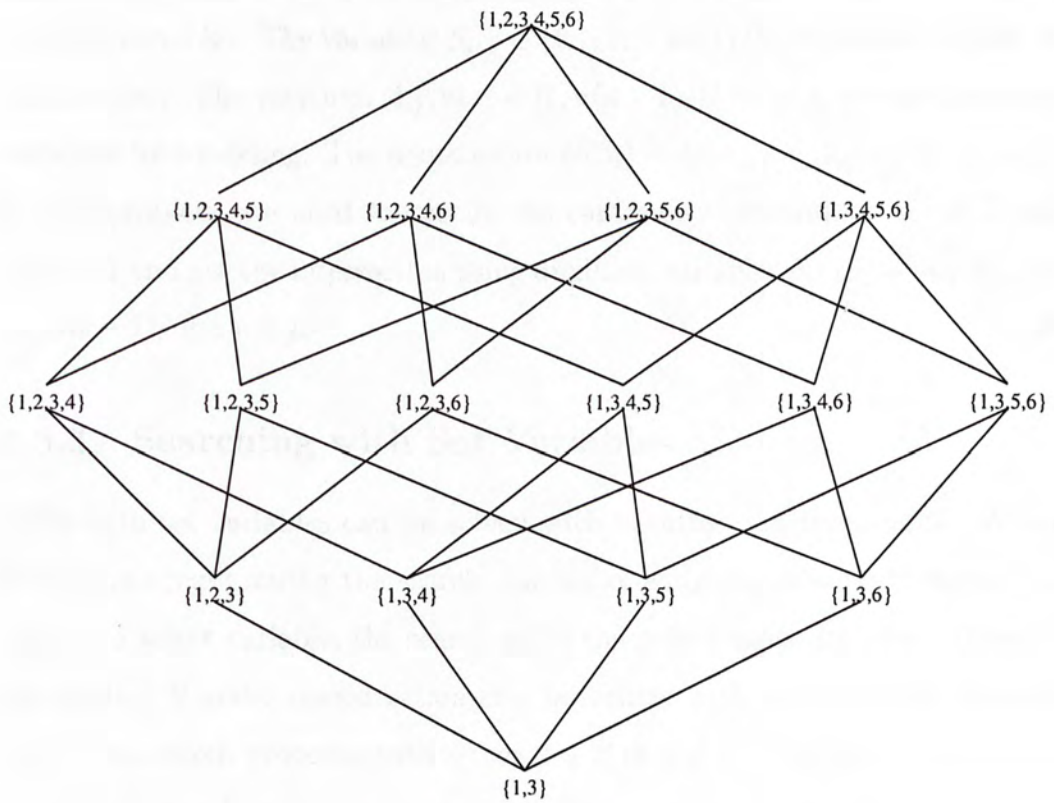


Figure 2.6: Set domain for the course selection of a student

In addition, there are cardinality constraints to restrict the cardinality of the sets.

**Example 2.6** *Steiner Triple System*

A steiner triple system of order  $n$  is to find a set of  $n(n-1)/6$  triples of distinct integer elements in  $\{1, \dots, n\}$  such that no two triples have more than one common element. We can model this problem with set variables. There are two kinds of variables. The variables  $S_i, i \in [1, \dots, n(n-1)/6]$ , represent the sets in the problem. The variables  $A_{ij}, \forall i, j \in [1, n(n-1)/6] \wedge i < j$ , are the auxiliary variables for modeling. The domains are  $D(S_i) = D(A_{ij}) = [\emptyset, \dots, \{1, \dots, n\}]$ . In this problem, we need to specify the cardinality constraints  $|S_i| = 3$  and  $|A_{ij}| \leq 1$  and get the intersection using auxiliary variables  $S_i \cap S_j = A_{ij}, \forall i, j \in [1, n(n-1)/6] \wedge i < j$ . ■

### 2.3.3 Searching with Set Variables

CSPs with set variables can be solved with backtracking tree search. When branching occurs during the search, instead of assigning a value from the domain of a select variable, the search splits the search space into two. Given a set variable  $S$  under consideration at a branching with a selected set element  $a \in S$ , the search proceeds with either  $a \in S$  or  $a \notin S$ . This splits the search space into two at each branching point. For example, a variable has domain  $[\emptyset, \{1, 2, 3\}]$  which is depicted in Figure 2.7(a). When we set  $3 \notin S$ , the set element 3 is removed from the possible set and the domain becomes  $[\emptyset, \{1, 2\}]$ . The modified domain is shown in Figure 2.7(b) in which the broken lines are connecting to the pruned set values. Figure 2.8 shows a complete search tree for variable  $S$  with domain  $D(S) = [\emptyset, \{1, 2, 3\}]$ . The current domain at every point of search tree is shown as a node. The search starts with the original domain. At each branching point, a set element  $a$  is picked from  $PS(S) \setminus RS(S)$



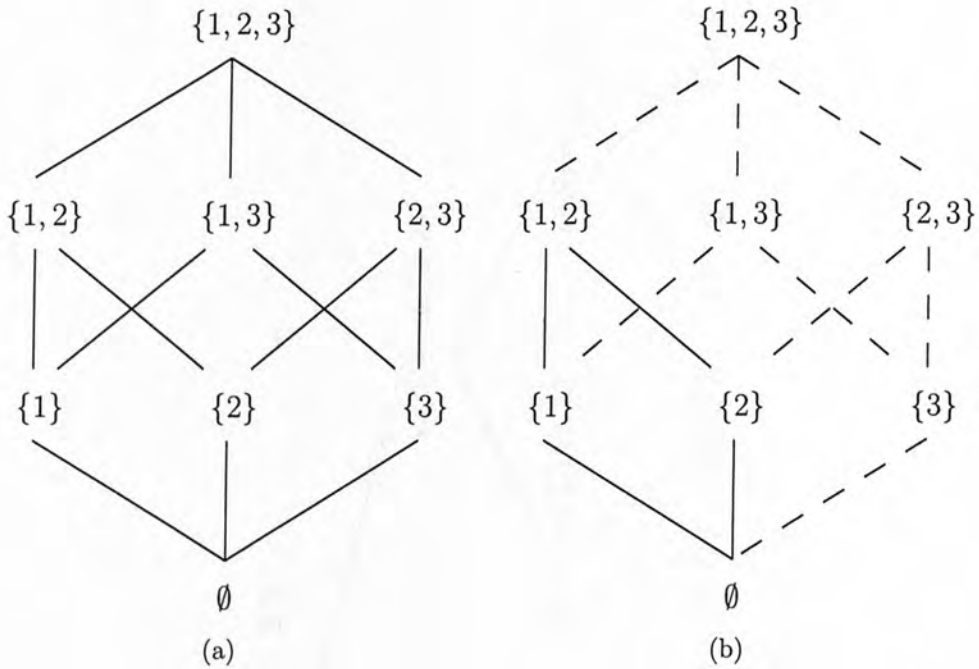


Figure 2.7: (a) An original domain for a set variable (b) A domain with 3 removed from the possible set

in lexicographical order. Each left branch is traversed with  $a \in S$  while right branch is traversed with  $a \notin S$ . Domain is narrowed during the tree traversal. In each leaf nodes, the domain contains a single element which is assigned to the variable  $S$ .

### 2.3.4 Set Bounds Consistency

Gervet [Ger97] defines local consistencies for set variables by reasoning the bounds of domain. We denote  $dom_S(C)$  all the values in the domain of set variable  $S$  that satisfy the constraint  $C$ .

**Definition 2.4** A set variable  $S$  with set interval domain  $[RS(S), PS(S)]$  is *set bounds consistent* with respect to a constraint  $C$  if and only if  $RS(S) = \bigcap dom_S(C) \wedge PS(S) = \bigcup dom_S(C)$ .

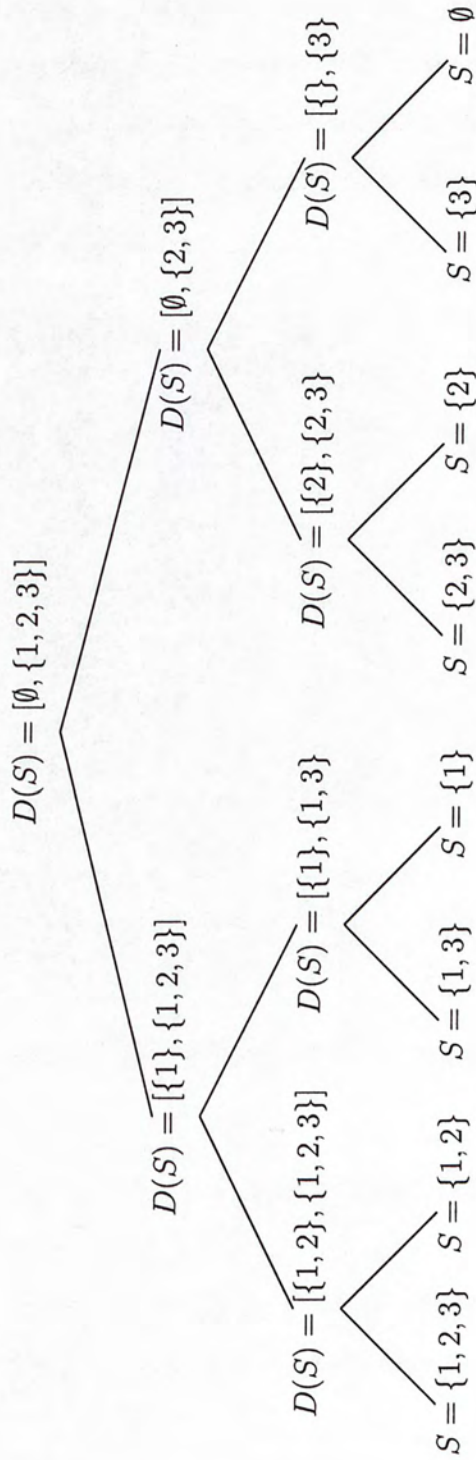


Figure 2.8: A complete search tree for a set variable

Set bounds consistency is enforced by applying *projection functions* as in [Ger97]. Each set constraint is associated with a set of projection functions which state how to modify the bounds of the set domains in the scope to maintain set bounds consistency. Projection functions of some common set constraints are listed in Table 2.1.

Constraint	Projection Functions
$S_1 \subseteq S_2$	$PS(S_1) \leftarrow PS(S_1) \cap PS(S_2)$ $RS(S_2) \leftarrow RS(S_2) \cup RS(S_1)$
$S_1 \cup S_2 = S_3$	$RS(S_1) \leftarrow RS(S_1) \cup RS(S_3) \setminus PS(S_2)$ $PS(S_1) \leftarrow PS(S_1) \cap PS(S_3)$ $RS(S_3) \leftarrow RS(S_3) \cup RS(S_1) \cup RS(S_2)$ $PS(S_3) \leftarrow PS(S_3) \cap PS(S_1) \cup PS(S_2)$
$S_1 \cap S_2 = S_3$	$RS(S_1) \leftarrow RS(S_1) \cup RS(S_3)$ $PS(S_1) \leftarrow PS(S_1) \setminus ((PS(S_1) \cap RS(S_2)) \setminus PS(S_3))$ $RS(S_3) \leftarrow RS(S_3) \cup RS(S_1) \cap RS(S_2)$ $PS(S_3) \leftarrow PS(S_3) \cap PS(S_1) \cap PS(S_2)$
$S_1 \setminus S_2 = S_3$	$RS(S_1) \leftarrow RS(S_1) \cup RS(S_3)$ $PS(S_1) \leftarrow PS(S_1) \setminus (PS(S_1) \setminus (PS(S_1) \setminus PS(S_2)))$ $RS(S_2) \leftarrow RS(S_2)$ $PS(S_2) \leftarrow PS(S_2) \setminus RS(S_3)$ $RS(S_3) \leftarrow RS(S_3) \cup RS(S_1) \setminus RS(S_2)$ $PS(S_3) \leftarrow PS(S_3) \cap PS(S_1) \setminus RS(S_2)$

Table 2.1: Projection functions for some common set constraints

**Example 2.7** Given a CSP with set variables  $S_1$  and  $S_2$ , with domains  $D(S_1) = [\{1, 2\}, \{1, 2, 3, 4, 5\}]$  and  $D(S_2) = [\emptyset, \{1, 2, 3, 4\}]$ . By considering the constraint  $S_1 \subseteq S_2$ , both set variables are not set bounds consistent. After enforcing set bounds consistency, the domains become  $D(S_1) = D(S_2) = [\{1, 2\}, \{1, 2, 3, 4\}]$ . ■



## Chapter 3

# Weighted Constraint

# Satisfaction with Set Variables

This chapter defines and introduces set variables in weighted constraint satisfaction problems. We discuss the issue on how to specify a set constraint by associating costs at the element level. We also show that the specification of set variables in weighted CSPs is a generalization of that in classical CSPs.

### 3.1 Set Variables

A set is a collection of distinct objects. When we describe a set with respect to a universal set, we are interested to know (1) what elements belong to the set, (2) what elements do not belong to the set, and (3) how many elements are in the set. From the first two points, we know the content of the set. The last point gives the *cardinality* of the set. A *set variable*  $S$  in WCSPs can only take a set value  $u$  from the domain  $D(S)$ . In this thesis, we consider integer sets. Thus, the domain of a set variable contains integer sets.

Each set variable  $S_i$  is associated with its universal set  $\mathcal{U}_i$  which contains all the possible elements in the set values in the domain. In other words, the universal set of a variable is the union over its initial domain  $\mathcal{U}_i = \bigcup D_0(S_i)$ .

The *universal set*  $\mathcal{U}$  of a WCSP is the union of all the universal sets associated with each set variable in the problem,  $\mathcal{U} = \bigcup_i \{\mathcal{U}_i\}$ .

Each set element in the universal set either exists or does not exist in the set variable. The *existence state* of a set element  $a$  with respect to a set variable  $S$  is  $a \in S$  which can be evaluated to a truth value from  $\{t, f\}$ . We denote  $E(S, a)$  as a set of truth values which contain the *possible existence states* of set element  $a$  for set variable  $S$ .

The *cardinality* of a set variable  $S$  is denoted as  $|S|$ . It is the value corresponding to the number of elements in the set value  $u$  when  $S$  is assigned with  $u$ .

## 3.2 Set Domains

The domain of a set variable is bounded by two sets, the possible set and required set, as in classical CSP. The possible set  $PS(S)$  of set variable  $S$  contains all the set elements which may be contained in the variable. The required set  $RS(S)$  of set variable  $S$  contains all the set elements which must exist in the variable. The possible set and required set are also called *lowest upper bound* and *greatest lower bound* respectively. The set domain is formed by  $D(S) = \{u | RS(S) \subseteq u \subseteq PS(S)\}$ . The set variables can only take the set values within the bounds inclusively. Initially, the range of possible cardinality of each set variable  $S$  is set to  $\{|RS(S)|, \dots, |PS(S)|\}$ , which is the maximum bounds for given  $RS(S)$  and  $PS(S)$ .

## 3.3 Set Constraints

In classical CSPs with set variables, we are deciding whether a particular set element should be contained in the set or not. In WCSPs, we associate the

costs for a particular set element to be contained in or excluded from the set value. When we assign  $\top$  to the existence (respectively inexistence) of a set element, we prohibit the set variable to contain (respectively remove) the set element. When the cost is less than  $\top$ , we allow the existence (respectively inexistence) of the set element with corresponding cost.

Set constraints defined here consider the existence state of each set element. This nature allows us to express the common soft set constraints which include element membership ( $a \in S_i, a \notin S_i$ ), equality ( $S_i = S_j$ ), subset ( $S_i \subseteq S_j$ ), union ( $S_i \cup S_j = S_k$ ), intersection ( $S_i \cap S_j = S_k$ ), difference ( $S_i \setminus S_j = S_k$ ), and cardinality ( $|S_i| = n, |S_i| \leq n, |S_i| \geq n$ ) where  $n$  is a constant. Complementation of two sets can be implemented using difference.

In this thesis, we focus on unary, binary, ternary, and cardinality constraints. These constraints enable us to express the common set constraints listed above with set variables. Since the performance of constraint propagation will degrade when the arity of a constraint is high, higher arity of constraints is usually decomposed to some primitive low arity constraints by introducing auxiliary variables [Cle87, Ger97]. For example, the constraint  $S_1 \cap S_2 \subseteq S_3 \cup S_4$  can be decomposed to  $S_1 \cap S_2 = A_1, S_3 \cup S_4 = A_2$  and  $A_1 \subseteq A_2$  with the introduction of two auxiliary set variables  $A_1$  and  $A_2$ . However, our definitions and algorithms do not restrict the arity of the constraints in theory.

Since the cost of a constraint is determined by the existence states of the set elements, we decompose the cost of the constraint by the corresponding *element cost functions* to reflect this relation. An element cost function maps the possible existence states in  $E(S, a)$  of a set element to an *element cost*. The cost for the constraint is the sum of all the element costs given from the set of element cost functions for that constraint. This approach gives



compact representation of constraints. Figure 3.1 shows the cases (a)  $2 \in S_1 \wedge 3 \notin S_2$ , (b)  $S_1 = S_2$ , and (c)  $S_1 \subseteq S_2$  as classical constraints in our representation. A dotted rectangle represent a set variable. Each oval in the rectangle is associated with a set element. The two circles in the oval represent the existence states of the set element and contains the corresponding unary costs. A circle drawn with broken line indicates an inconsistent existence state which is removed from the set of possible existence states  $E$  of corresponding set variable and set element. The binary costs between two set variables are indicated on the lines representing constraints.

### 3.3.1 Zero-arity Constraint

As in WCSPs, there is a zero-arity constraint  $C_\emptyset$  in the problem. The cost of the zero-arity constraint can be interpreted as the global lower bound of the problem. The problem contains no solutions when  $C_\emptyset = \top$ .

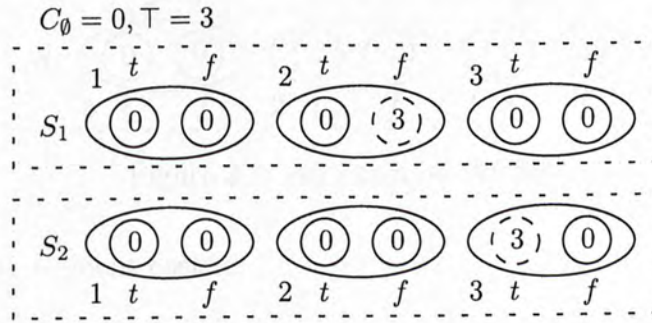
### 3.3.2 Unary Constraints

A unary constraint  $C_i$  assigns costs to assignments to variable  $S_i$  ( $C_i : D(S_i) \rightarrow [0, \dots, k]$ ). The corresponding *unary element cost function*, which assigns costs for the existence ( $t$ ) and inexistence ( $f$ ) for each set element  $a \in \mathcal{U}_i$  with respect to set variable  $S_i$ , is  $\varphi_{(i)/a} : \{t, f\} \rightarrow [0, \dots, k]$ . The unary cost of set value  $u$  for variable  $S_i$  is for constraint  $C_i$  is defined as  $C_i(u) = \sum_{a \in \mathcal{U}_i} \varphi_{(i)/a}(a \in u)$ .

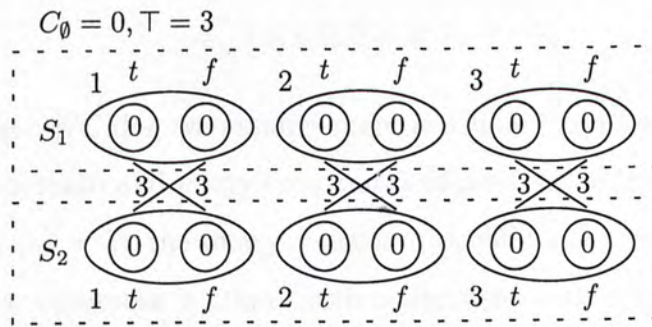
In the WCSP in Figure 3.2(a), for example, the cost of  $2 \in S_1$  is 0 while that of  $2 \notin S_1$  is 3 :

$$\varphi_{(1)/2}(\alpha) = \begin{cases} 0 & \text{if } \alpha = t; \\ 3 & \text{if } \alpha = f. \end{cases}$$

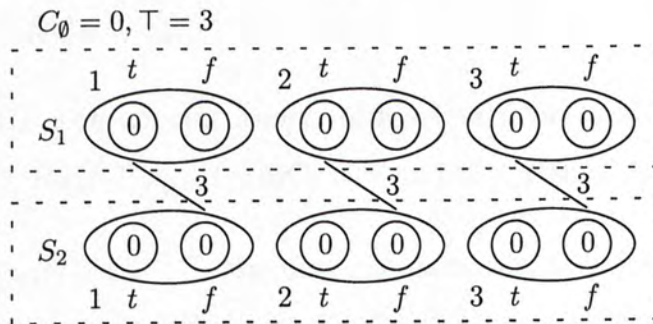
The cost  $C_1(\{1, 2\})$  for the unary constraint  $C_1$  on  $\{1, 2\}$  equals the sum



(a)



(b)



(c)

Figure 3.1: (a)  $2 \in S_1 \wedge 3 \notin S_2$  (b)  $S_1 = S_2$  (c)  $S_1 \subseteq S_2$

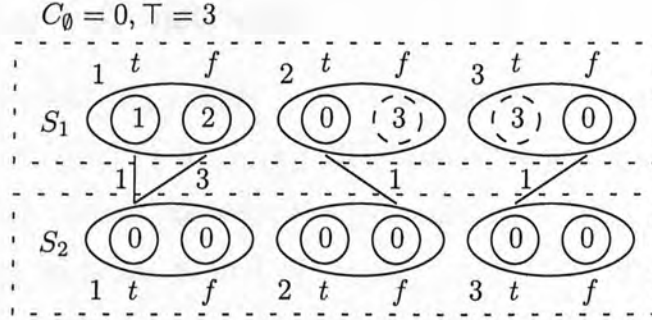


Figure 3.2: An example WCSP

of all the unary element costs :

$$\begin{aligned} C_1(\{1, 2\}) &= \varphi_{(1)/1}(t) \oplus \varphi_{(1)/2}(t) \oplus \varphi_{(1)/3}(f) \\ &= 1 \oplus 0 \oplus 0 = 1 \end{aligned}$$

As in integer WCSPs, we assume there is a unary constraint for each set variable. The domain and unary constraints of a set variable are interchangeable. When  $C_i(u) = \top$ , the unary constraint prohibits the variable  $S_i$  taking the set value  $u$ ; otherwise it allows such assignment with cost  $C_i(u)$ . As reasoning on each domain value for a set variable is impractical, we focus on the bounds of a domain.

**Definition 3.1** The *domain bounds* of a set variable  $S_i$  is  $[RS(S_i), PS(S_i)]$  such that  $\forall a \in RS(S_i), \varphi_{(i)/a}(f) \oplus C_\emptyset = \top$  and  $\forall b \in PS(S_i), \varphi_{(i)/b}(t) \oplus C_\emptyset < \top$ .

When  $\varphi_{(i)/a}(t) \oplus C_\emptyset < \top$ , the unary element cost function allows the existence of the set element  $a$  in the set  $S_i$  with the corresponding cost. Therefore the set element  $a$  can exist in the set. When  $\varphi_{(i)/a}(f) \oplus C_\emptyset = \top$ , the unary element cost function forbids the inexistence of the set element  $a$ , and the set element  $a$  must exist in the set. According to the unary constraint  $C_i$  in Figure 3.2(a), we have  $PS(S_i) = \{1, 2\}$  and  $RS(S_i) = \{2\}$  since  $C_\emptyset = 0$  and  $\top = 3$ .



### 3.3.3 Binary Constraints

A binary constraint  $C_{i,j}$  assigns costs to assignments to variables  $S_i$  and  $S_j$  ( $C_{i,j} : D(S_i) \times D(S_j) \rightarrow [0, \dots, k]$ ). The corresponding *binary element cost function*, which assigns costs for the existence states of a set element  $a \in \mathcal{U}_i \cup \mathcal{U}_j$  for set variables  $S_i$  and  $S_j$ , is  $\varphi_{(i,j)/a} : \{t, f\} \times \{t, f\} \rightarrow [0, \dots, k]$ . Since  $\mathcal{U}_i$  may not be equal to  $\mathcal{U}_j$ ,  $\forall \alpha, \beta \in \{t, f\}$ , the binary element cost function  $\varphi_{(i,j)/a}(t, \alpha) = \top, \forall a \notin \mathcal{U}_i$  and  $\varphi_{(i,j)/a}(\beta, t) = \top, \forall a \notin \mathcal{U}_j$ . The binary constraint of the set variables  $S_i$  and  $S_j$  can be defined as :  $C_{ij}(u, v) = \sum_{a \in \mathcal{U}_i \cup \mathcal{U}_j} \varphi_{(i,j)/a}(a \in u, a \in v)$ .

Figure 3.2(a) shows the binary element costs among the set elements of  $S_1$  and  $S_2$ . The costs are indicated on the lines linking the existence states of the elements in the two sets. No lines are drawn if the cost is 0. According to the figure, the element cost for 1 in  $S_1$  and  $S_2$  is :

$$\varphi_{(1,2)/1}(\alpha, \beta) = \begin{cases} 1 & \text{if } \alpha = t \wedge \beta = t; \\ 3 & \text{if } \alpha = f \wedge \beta = t; \\ 0 & \text{otherwise} \end{cases}$$

The binary cost for  $S_1 = \{1, 2, 3\}$  and  $S_2 = \{1, 3\}$  is the sum of all the binary element costs :

$$\begin{aligned} C_{12}(\{1, 2, 3\}, \{1, 3\}) \\ &= \varphi_{(1,2)/1}(t, t) \oplus \varphi_{(1,2)/2}(t, f) \oplus \varphi_{(1,2)/3}(t, t) \\ &= 1 \oplus 1 \oplus 0 = 2 \end{aligned}$$

### 3.3.4 Ternary Constraints

A ternary constraint  $C_{i,j,k}$  assigns costs to assignments to variables  $S_i$ ,  $S_j$  and  $S_k$  ( $C_{i,j,k} : D(S_i) \times D(S_j) \times D(S_k) \rightarrow [0, \dots, k]$ ). The corresponding *ternary element cost function*, which assigns costs to the existence states of a

set element  $a \in \mathcal{U}_i \cup \mathcal{U}_j \cup \mathcal{U}_k$  for variables  $S_i$ ,  $S_j$  and  $S_k$ , is  $\varphi_{(i,j,k)/a} : \{t, f\} \times \{t, f\} \times \{t, f\} \rightarrow [0, \dots, k]$ .

Similar to the case for the binary constraint, there may be an element  $a \in \mathcal{U}_i \cup \mathcal{U}_j \cup \mathcal{U}_k$  where  $a \notin \mathcal{U}_i$ . In this case, all the ternary element cost functions taking  $a \in S_i$  return  $\top$  as the cost. This is also the same for variables  $S_j$  and  $S_k$ . The ternary constraint of the set variables  $S_i$ ,  $S_j$  and  $S_k$  can be defined as :  $C_{i,j,k}(u, v, w) = \sum_{a \in \mathcal{U}_i \cup \mathcal{U}_j \cup \mathcal{U}_k} \varphi_{(i,j,k)/a}(a \in u, a \in v, a \in w)$ .

### 3.3.5 Cardinality Constraints

A cardinality constraint  $C_{|i|}$  assigns costs to assignments to a set variable  $S_i$  according to the cardinality of  $S_i$ . It is decomposed as  $C_{|i|} = (Cost_{|i|} \circ Card)$  where  $Card : D(S_i) \rightarrow \mathbb{N} \cup \{0\}$  and  $Cost_{|i|} : \mathbb{N} \cup \{0\} \rightarrow [0, \dots, k]$ . This constraint first maps the assignment of the variable  $S_i$  to its cardinality  $|S_i|$  by using  $Card$ . It then assigns costs to  $|S_i|$  by  $Cost_{|i|}$ .

## 3.4 Characteristics

### 3.4.1 Space Complexity

The space complexity of constraint representation is greatly reduced with our proposal. Table 3.1 tabulates the storage requirement for unary, binary, and ternary set constraints in terms of number of costs specified when set variables and integer variables are used. When we use integer variables to simulate set variables, each set value in the set domain is mapped to an integer in the integer domain. Thus, the domain size and space complexity of constraints for integer variables grow exponentially with the number of set elements in the sets. Set constraint specification is compact in our proposal in which the space complexity is linear to the number of set elements in the sets.

Arity of Constraint	Space Complexity	
	Set Variables	Integer Variables
Unary	$2e$	$2^e$
Binary	$4e$	$2^{2e}$
Ternary	$8e$	$2^{3e}$

Table 3.1: Space complexity of set constraints of different arities with the use of set variables and integer variables ( $e$  is the maximum number of set elements in the sets)

### 3.4.2 Generalization

**Property 3.1** The classical versions of element membership, equality, subset, union, intersection, difference, and cardinality constraints can be modeled in our WCSP framework with element costs 0 and  $\top$ .

**Proof** Cost functions, with costs 0 and  $\top$  only, for the classical versions of element membership, equality, subset, union, intersection, difference, and cardinality constraints are listed in Table 3.2.

□

**Definition 3.2** Given a classical CSP  $\mathcal{P}_C = (\mathcal{X}_C, \mathcal{D}_C, \mathcal{C}_C)$  and a WCSP  $\mathcal{P}_W = (k, \mathcal{X}_W, \mathcal{D}_W, \mathcal{C}_W)$ ,  $\mathcal{P}_C$  and  $\mathcal{P}_W$  are *equivalent* to each other if and only if  $\mathcal{X}_C \equiv \mathcal{X}_W$  and for each complete assignment in the problem  $t$ ,  $\mathcal{V}(t) = 0$  in  $P_W$  if and only if  $t$  is a solution of  $P_C$ .

**Property 3.2** When a WCSP with set variables involves only 0 and  $\top$  in the element costs, the WCSP can be transformed into an equivalent problem with classical set constraints only.

**Proof** Since set constraints in a WCSP are defined with corresponding element cost functions, this property can be shown by transforming every element



Set Constraint	Equivalent Cost Function
$a \in S_i$	$\varphi_{(i)/a}(\alpha) = \begin{cases} \top & \text{if } \alpha = f; \\ 0 & \text{otherwise.} \end{cases}$
$a \notin S_i$	$\varphi_{(i)/a}(\alpha) = \begin{cases} \top & \text{if } \alpha = t; \\ 0 & \text{otherwise.} \end{cases}$
$S_i = S_j$	$\forall a \in \mathcal{U}_i \cup \mathcal{U}_j, \varphi_{(i,j)/a}(\alpha, \beta) = \begin{cases} \top & \text{if } \alpha \neq \beta; \\ 0 & \text{otherwise.} \end{cases}$
$S_i \subseteq S_j$	$\forall a \in \mathcal{U}_i \cup \mathcal{U}_j, \varphi_{(i,j)/a}(\alpha, \beta) = \begin{cases} \top & \text{if } \alpha = t \wedge \beta = f; \\ 0 & \text{otherwise.} \end{cases}$
$S_i \cup S_j = S_k$	$\forall a \in \mathcal{U}_i \cup \mathcal{U}_j \cup \mathcal{U}_k, \varphi_{(i,j,k)/a}(\alpha, \beta, \gamma) = \begin{cases} \top & \text{if } (\alpha = t \vee \beta = t) \wedge \gamma = f; \\ \top & \text{if } \alpha = f \wedge \beta = f \wedge \gamma = t; \\ 0 & \text{otherwise.} \end{cases}$
$S_i \cap S_j = S_k$	$\forall a \in \mathcal{U}_i \cup \mathcal{U}_j \cup \mathcal{U}_k, \varphi_{(i,j,k)/a}(\alpha, \beta, \gamma) = \begin{cases} \top & \text{if } (\alpha = f \vee \beta = f) \wedge \gamma = t; \\ \top & \text{if } \alpha = t \wedge \beta = t \wedge \gamma = f; \\ 0 & \text{otherwise.} \end{cases}$
$S_i \setminus S_j = S_k$	$\forall a \in \mathcal{U}_i \cup \mathcal{U}_j \cup \mathcal{U}_k, \varphi_{(i,j,k)/a}(\alpha, \beta, \gamma) = \begin{cases} \top & \text{if } \beta = t \wedge \gamma = t; \\ \top & \text{if } \alpha = t \wedge \beta = f \wedge \gamma = f; \\ \top & \text{if } \alpha = f \wedge \beta = f \wedge \gamma = t; \\ 0 & \text{otherwise.} \end{cases}$
$ S_i  = n$	$C_{ i } = \begin{cases} \top & \text{if }  S_i  \neq n; \\ 0 & \text{otherwise.} \end{cases}$
$ S_i  \leq n$	$C_{ i } = \begin{cases} \top & \text{if }  S_i  > n; \\ 0 & \text{otherwise.} \end{cases}$
$ S_i  \geq n$	$C_{ i } = \begin{cases} \top & \text{if }  S_i  < n; \\ 0 & \text{otherwise.} \end{cases}$

Table 3.2: Soft versions of common classical set constraints

cost function to an equivalent classical membership constraint. For each unary element cost function,  $\varphi_{(i)/a}(t) = \top$  becomes  $a \notin S_i$  and  $\varphi_{(i)/a}(f) = \top$  becomes  $a \in S_i$ . The transformations for binary element cost functions are listed below.

Cost Function	Classical Constraint
$\varphi_{(i,j)/a}(t, t) = \top$	$a \notin (S_i \cap S_j)$
$\varphi_{(i,j)/a}(t, f) = \top$	$a \notin (S_i \setminus S_j)$
$\varphi_{(i,j)/a}(f, t) = \top$	$a \notin (S_j \setminus S_i)$
$\varphi_{(i,j)/a}(f, f) = \top$	$a \in (S_i \cup S_j)$

The transformations for ternary element cost functions are similar. For the cardinality constraints, we transform each  $n$  such that  $Cost_{|i|}(n) = \top$  to  $|S_i| \neq n$  as a classical constraint.  $\square$

**Theorem 3.1** WCSPs with set variables subsumes classical CSPs with set variables.

**Proof** This follows directly from Properties 3.1 and 3.2.  $\square$

## Chapter 4

# Consistency Notions and Algorithms for Set Variables

This chapter defines some local consistency notions applied to WCSPs with set variables. Examples are given to illustrate the concepts. We also give algorithms to enforce these local consistencies for set variables and constraints. We give the complexity and prove the correctness of the algorithms.

### 4.1 Consistency Notions

The costs of a set constraint are specified at the element level via element cost functions. We can define local consistencies for the element cost functions as follows.

#### 4.1.1 Element Node Consistency

**Definition 4.1** An existence state  $\alpha$  of set element  $a$  is *element node consistent (ENC)* with respect to unary constraint  $C_i$  if  $C_\emptyset \oplus \varphi_{(i)/a}(\alpha) < \top$ . A set element  $a$  is *ENC* if



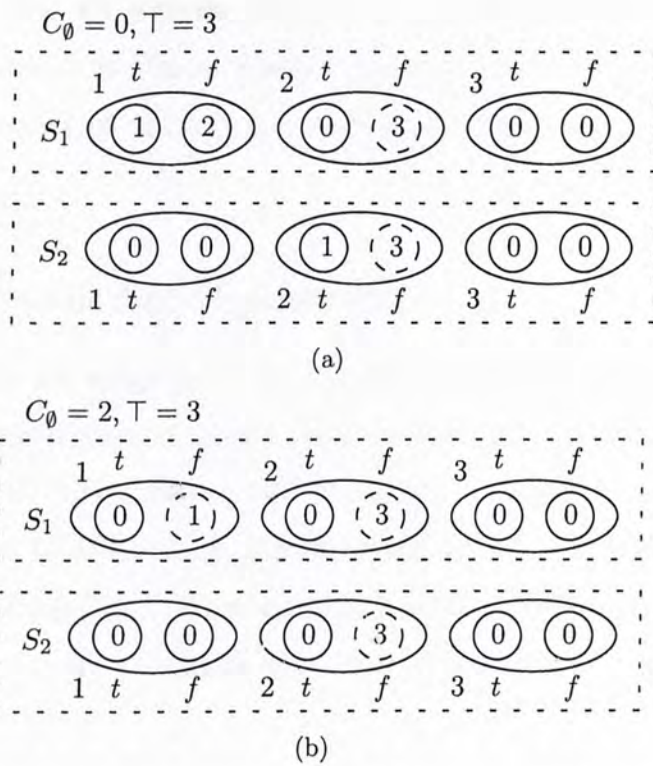


Figure 4.1: (a) A WCSP which is not ENC (b) An equivalent WCSP which is ENC

1. all its possible existence states in  $E(S_i, a)$  are ENC with respect to unary constraint  $C_i$ , and
2.  $\exists \alpha \in E(S_i, a)$  such that  $\varphi_{(i)/a}(\alpha) = 0$ .

The existence state  $\alpha$  is a support for the set element  $a$ . A set variable  $S_i$  is *ENC* with respect to unary constraint  $C_i$  if every set element is ENC. A WCSP is *ENC* if every set variable  $S_i$  is ENC.

**Example 4.1** Figure 4.1(a) shows a WCSP with set variables, which is not ENC since the set elements 1 in  $S_1$  and 2 in  $S_2$  are not ENC. The minimum cost for the existence states of 1 in  $S_1$  is 1. For 2 in  $S_2$ , where  $E(S_2, 2) = \{t\}$ , the only possible existence state,  $t$ , costs 1, which is also not 0. An equivalent

WCSP is obtained if 1 is subtracted from the costs for set element 1 in  $S_1$  and, at the same time, 1 is also subtracted from the cost for set element 2 in  $S_2$ . This contributes a cost of 2 to the global lower bound,  $C_\emptyset$ . The result is shown in Figure 4.1(b). ■

### 4.1.2 Element Arc Consistency

**Definition 4.2** An existence state  $\alpha$  of set element  $a$  is *element arc consistent (EAC)* with respect to binary constraint  $C_{i,j}$  if  $\exists \beta \in E(S_j, a)$  such that  $\varphi_{(i,j)/a}(\alpha, \beta) = 0$ . An existence state  $\beta$  is a support of the existence state  $\alpha$ . A set element is *EAC* if all its possible existence states are EAC with respect to the binary constraint  $C_{i,j}$ . A set variable is *EAC* if every set element is EAC with respect to binary constraint  $C_{i,j}$ . A WCSP is *EAC* if every set variable is EAC and ENC.

**Example 4.2** Figure 4.2(a) shows a WCSP with set variables, which is not EAC since the existence state for  $1 \in S_1$  has no support in  $S_2$ . The existence state for  $2 \in S_2$  is not EAC because the binary cost associated with the only existence state  $t$  is 1. Figure 4.2(b) shows an equivalent WCSP which is EAC. The minimum binary cost for  $1 \in S_1$  is subtracted from the binary constraint and added to the unary cost of  $1 \in S_1$ . Similarly, the binary cost  $\varphi_{(1,2)/2}(t, t)$  is sent to the unary cost of  $2 \in S_2$ . ■

### 4.1.3 Element Hyper-arc Consistency

**Definition 4.3** An existence state  $\alpha$  of set element  $a$  is *element hyper-arc consistent (EHAC)* with respect to ternary constraint  $C_{i,j,k}$  if  $\exists \beta \in E(S_j, a), \gamma \in E(S_k, a)$  such that  $\varphi_{(i,j,k)/a}(\alpha, \beta, \gamma) = 0$ . Existence states  $\beta$  and  $\gamma$  are supports of the existence state  $\alpha$ . The set element is *EHAC* if all its possible existence

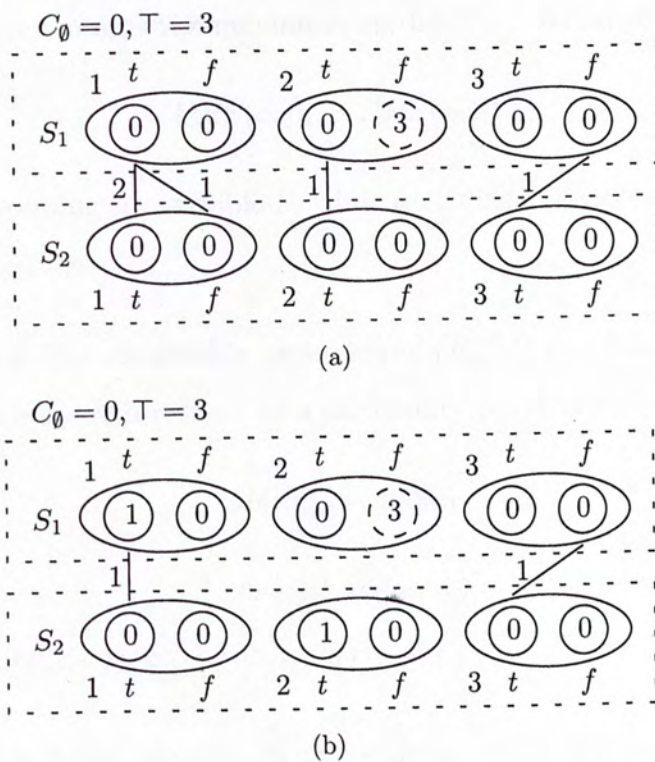


Figure 4.2: (a) A WCSP which is not EAC (b) An equivalent WCSP which is EAC



states are EHAC with respect to ternary constraint  $C_{i,j,k}$ . A set variable is *EHAC* if every set element is EHAC with respect to ternary constraint  $C_{i,j,k}$ . A WCSP is *EHAC* if every set variable is EHAC and ENC.

#### 4.1.4 Weighted Cardinality Consistency

For the cardinality constraint, we adopt a notion of weighted cardinality consistency which maintains the maximum cardinality interval within

$$\{|RS(S_i)|, \dots, |PS(S_i)|\}$$

for the corresponding set variable  $S_i$  while removing the inconsistent cardinality from the bounds.

**Definition 4.4** The *cardinality upper bound*  $ub(|S_i|)$  and *lower bound*  $lb(|S_i|)$  of a set variable  $S_i$  with respect to a cardinality constraint  $C_{|i|}$  are defined as

$$ub(|S_i|) = \max A$$

$$lb(|S_i|) = \min A$$

where  $A = \{|u| \mid u \in D(S_i) \wedge C_{|i|}(u) \oplus C_\emptyset < \top\}$ .

**Definition 4.5** A set variable  $S_i$  is *weighted cardinality consistent (WCC)* with respect to a cardinality constraint  $C_{|i|}$  if

1. the cardinality upper bound  $ub(|S_i|) \leq |PS(S_i)|$  and the cardinality lower bound  $lb(|S_i|) \geq |RS(S_i)|$ ,
2.  $Cost_{|i|}(lb(|S_i|)) \oplus C_\emptyset < \top$ , and
3.  $Cost_{|i|}(ub(|S_i|)) \oplus C_\emptyset < \top$ .

### 4.1.5 Weighted Set Bounds Consistency

As in classical set CSPs, we do not reason about each domain value in the set domain due to its high complexity. Instead, we enforce the consistency by adjusting the bounds of a set domain. Since set constraints in WCSPs are defined in terms of sum of element cost functions, we reason on the bounds of a set domain by considering the cost for the existence of each set element.

Given a set variable  $S$  and a constraint  $C$ , we denote by  $wdom_S(C)$  a set containing all set values  $u$  such that

1.  $RS(S) \subseteq u \subseteq PS(S)$ , and
2.  $\exists t$ , such that  $\{S \mapsto u\} \subseteq t$  and  $C(t) \oplus C_\emptyset < \top$ .

**Definition 4.6** A set variable  $S$  with domain  $[RS(S), PS(S)]$  is *weighted set bounds consistent (WSBC)* with respect to a constraint  $C$  if and only if  $RS(S) = \bigcap wdom_S(C) \wedge PS(S) = \bigcup wdom_S(C)$ .

**Theorem 4.1** A set variable  $S$  is *WSBC* with respect to a unary constraint  $C_i$  (or a binary constraint  $C_{i,j}$  or a ternary constraint  $C_{i,j,k}$ ) if  $S$  is ENC with respect to  $C_i$  (or EAC with respect to  $C_{i,j}$  or EHAC with respect to  $C_{i,j,k}$ ).

**Proof** By the definition of  $RS(S)$  and  $PS(S)$  for set variable  $S$ , it is trivial to show that any set element in  $RS(S)$  must exist and any set element not in  $PS(S)$  must not exist. The following proves that no extra elements can be put in  $RS(S)$  or taken out from  $PS(S)$  when ENC (or EAC or EHAC) is enforced.

For unary constraint, suppose  $\exists a \notin RS(S_i)$  such that  $\forall u \in D(S_i), a \notin u \rightarrow C(u) \oplus C_\emptyset = \top$ . When  $S_i$  is ENC,  $a \notin RS(S_i)$  implies  $C_\emptyset \oplus \varphi_{(i)/a}(f) < \top$ . We can always construct a set value  $v$  for  $S_i$  such that  $C_i(v) = 0$ . Now, we set  $a \notin S_i$  and form new a set value  $w$  with cost  $\varphi_{(i)/a}(f)$ , then  $C_i(w) \oplus C_\emptyset < \top$  leads to contradiction. For binary (or ternary) constraint, when  $S_i$  is EAC (or



EHAC) we can find a support for the set value  $w'$  for  $S_i$  where  $a \notin w'$  with cost 0 with respect to the binary (ternary) constraint. Therefore,  $a \notin RS(S_i)$ .

On the other hand, suppose  $\exists a \in PS(S_i)$  such that  $\forall u \in D(S_i), a \in u \rightarrow C(u) \oplus C_\emptyset = \top$ . Since  $a \in PS(S_i)$ ,  $C_\emptyset \oplus \varphi_{(i)/a}(t) < \top$ . We can always construct a set value  $v$  for  $S_i$  such that  $C_i(v) = 0$ . Now we set  $a \in S_i$  and form a set value  $w$  with cost  $\varphi_{(i)/a}(t)$ , then  $C_i(w) \oplus C_\emptyset < \top$  leads to contradiction. Similar to the above case, we can find a support for  $w'$  for  $S_i$  where  $a \in w'$  with cost 0 with respect to the binary (ternary) constraint. Therefore,  $a \in PS(S_i)$ .  $\square$

**Theorem 4.2** When a WCSP with set variables involves costs 0 and  $\top$  only, WSBC = SBC.

**Proof** By the definition of  $RS(S)$  and  $PS(S)$  for set variable  $S$ , since  $\forall a \in RS(S), C_\emptyset \oplus \varphi_{(i)/a}(f) = \top$ , any set value must contain element  $a$ . In addition, since  $\forall a \notin PS(S), C_\emptyset \oplus \varphi_{(i)/a}(t) = \top$ , any set value must not contain  $a$ . According to Theorem 4.1, WSBC ensures that each set element  $a \in PS(S) \setminus RS(S)$  can be extended to form a set value with cost  $C_\emptyset \oplus \varphi_{(i)/a}(t) < \top$ . Since there are costs 0 and  $\top$  only,  $C_\emptyset \oplus \varphi_{(i)/a}(t) = 0$  which implies that the set element  $a$  can be contained in the set value.  $\square$

## 4.2 Consistency Enforcing Algorithms

The element consistencies can be enforced in a similar way as in enforcing local consistencies in integer WCSPs. The enforcement procedures involve sending costs from ternary and binary constraints to unary constraints and from unary constraints to global lower bound  $C_\emptyset$  to obtain a support. The costs subtracted from the constraints are added to the appropriate location to preserve equivalence.



### 4.2.1 Enforcing Element Node Consistency

Algorithm 4.1 shows the procedure for enforcing ENC. The function  $\text{ENC}()$  involves two major steps. The first step forces unary support for each set element  $a$  in the variable universe  $\mathcal{U}_i$  for each set variable  $S_i$  in  $\text{FindUnarySupports}()$ . A minimum cost of  $\varphi_{(i)/a}$  among the possible existence is determined. This cost is added to the global lower bound  $C_\emptyset$  and subtracted from the unary element cost function  $\varphi_{(i)/a}$ . In the second step, the domain of each set variable is narrowed in  $\text{PruneVar}()$ . A set element  $a$  for set variable  $S_i$  is removed from  $PS(S_i)$  if  $\varphi_{(i)/a}(t) \oplus C_\emptyset = \top$  or included in  $RS(S_i)$  if  $\varphi_{(i)/a}(f) \oplus C_\emptyset = \top$ .

For each set element of set variable,  $\text{FindUnarySupports}()$  and  $\text{PruneVar}()$  both have complexity  $\mathcal{O}(1)$  as each set element has maximum two existence states. Therefore, given a WCSP with  $n$  set variables and each with maximum  $e$  set elements, the procedure  $\text{ENC}()$  has complexity  $\mathcal{O}(ne + ne) = \mathcal{O}(ne)$ .

**Theorem 4.3** Given a WCSP  $P$ , Algorithm 4.1 transforms  $P$  to  $P'$  such that

1.  $P'$  is equivalent to  $P$ , and
2.  $P'$  is ENC.

**Proof** The procedure given in Algorithm 4.1 only involves basic operations on the costs. In  $\text{FindUnarySupports}()$ , the minimum cost of  $\varphi_{(i)/a}$  is added to  $C_\emptyset$ . At the same time, the same amount of cost is subtracted from  $\varphi_{(i)/a}$  for all possible existence states. An equivalent on cost evaluation is preserved.

The cost operations in  $\text{FindUnarySupports}()$  ensure that there is a unary support for each set element of a set variable. Inconsistent existence states are pruned in  $\text{PruneVar}()$ . Thus, the transformed problem is ENC.  $\square$

---

**Algorithm 4.1:** Enforcing element node consistency
 

---

```

1 Procedure FindUnarySupports( $S_i, a$ )
2 begin
3    $c := \min_{\alpha \in E(S_i, a)} (\varphi_{(i)/a}(\alpha))$ 
4    $C_\emptyset := C_\emptyset \oplus c$ 
5   for  $\alpha \in E(S_i, a)$  do
6      $\lfloor \varphi_{(i)/a}(\alpha) := \varphi_{(i)/a}(\alpha) \ominus c$ 
7 end

8 Procedure PruneVar( $S_i, a$ )
9 begin
10   $\text{change} := \text{false}$ 
11  if  $\varphi_{(i)/a}(t) \oplus C_\emptyset = \top$  then
12     $PS(S_i) := PS(S_i) \setminus \{a\}$ 
13     $\lfloor \text{change} := \text{true}$ 
14  if  $\varphi_{(i)/a}(f) \oplus C_\emptyset = \top$  then
15     $RS(S_i) := RS(S_i) \cup \{a\}$ 
16     $\lfloor \text{change} := \text{true}$ 
17  return  $\text{change}$ 
18 end

19 Procedure ENC( $\mathcal{X}, \mathcal{D}, C$ )
20 begin
21  for  $S_i \in \mathcal{X}$  do
22    for  $a \in \mathcal{U}_i$  do
23       $\lfloor$  FindUnarySupports( $S_i, a$ )
24  for  $S_i \in \mathcal{X}$  do
25    for  $a \in \mathcal{U}_i$  do
26       $\lfloor$  PruneVar( $S_i, a$ )
27 end

```

---

---

**Algorithm 4.2:** Enforcing element arc consistency

---

```

1 Procedure FindBinarySupports( $S_i, S_j, a$ )
2 begin
3   for  $\beta \in E(S_j, a)$  do
4      $c := \min_{\alpha \in E(S_i, a)} (\varphi_{(i,j)/a}(\alpha, \beta))$ 
5      $\varphi_{(j)/a}(\beta) := \varphi_{(j)/a}(\beta) \oplus c$ 
6     for  $\alpha \in E(S_i, a)$  do
7        $\varphi_{(i,j)/a}(\alpha, \beta) := \varphi_{(i,j)/a}(\alpha, \beta) \ominus c$ 
8   FindUnarySupports( $S_j, a$ )
9 end

10 Procedure EAC( $\mathcal{X}, \mathcal{D}, \mathcal{C}$ )
11 begin
12    $Q := \emptyset$ 
13   for  $S_i \in \mathcal{X}$  do
14     for  $a \in \mathcal{U}_i$  do
15        $Q := Q \cup \{(S_i, a)\}$ 
16   while  $Q \neq \emptyset$  do
17      $(S_i, a) \in Q$ 
18      $Q := Q \setminus \{(S_i, a)\}$ 
19     for  $C_{i,j} \in \mathcal{C}$  do
20       FindBinarySupports( $S_i, S_j, a$ )
21     for  $S_m \in \mathcal{X}$  do
22       for  $a \in \mathcal{U}_m$  do
23         if PruneVar( $S_m, a$ ) then
24            $Q := Q \cup \{(S_m, a)\}$ 
25 end

```

---



### 4.2.2 Enforcing Element Arc Consistency

The procedure for enforcing EAC is depicted in Algorithm 4.2. The function  $\text{EAC}()$  computes and stores in  $Q$  the set of all possible pairs of set variable and set element. Each time a pair  $(S_i, a)$  is picked out from  $Q$ . The algorithm finds binary supports for  $a \in S_j$  with constraints  $C_{i,j} \in \mathcal{C}$  in  $\text{FindBinarySupports}()$ . A binary support for each existence state  $\beta$  of set element  $a$  of set variable  $S_j$  is forced by sending minimum cost of  $\varphi_{(i,j)/a}(\alpha, \beta)$  to the unary cost  $\varphi_{(j)/a}(\beta)$ . The cost is also subtracted from the binary costs  $\varphi_{(i,j)/a}(\alpha, \beta)$ . Since the unary cost  $\varphi_{(j)/a}(\beta)$  and  $a \in S_j$  may not be ENC,  $\text{FindUnarySupports}()$  is called to force ENC. In the end of each iteration, all set elements in each set variable are checked to prune any inconsistent existence state in  $\text{PruneVar}()$ .

The procedure  $\text{FindBinarySupports}()$  has complexity  $\mathcal{O}(1)$  as each set element has a maximum of two existence states. Given a WCSP with  $n$  set variables and each with maximum  $e$  set elements. In  $\text{EAC}()$ , each pair of set variable and set element can be re-inserted into  $Q$  once. The complexity of  $\text{EAC}()$  is thus  $\mathcal{O}(ne + (ne)(n + ne)) = \mathcal{O}(n^2e^2)$ .

**Theorem 4.4** Given a WCSP  $P$ , Algorithm 4.2 transforms  $P$  to  $P'$  such that

1.  $P'$  is equivalent to  $P$ , and
2.  $P'$  is EAC.

**Proof** The procedure given in Algorithm 4.2 only involves basic operations on the costs. In  $\text{FindBinarySupports}()$ , for each existence state  $\beta \in E(S_j, a)$ , the minimum cost of  $\varphi_{(i,j)/a}(\alpha, \beta)$  for  $\alpha \in E(S_i, a)$  is added to  $\varphi_{(j)/a}(\beta)$ . At the same time, the same amount of cost is subtracted from  $\varphi_{(i,j)/a}(\alpha, \beta)$  for all  $\alpha \in E(S_i, a)$ . An equivalent on cost evaluation is preserved.

The cost operations in `FindBinarySupports()` ensure that there is a binary support for each existence state of a set element for a set variable. Inconsistent existence states are pruned in `PruneVar()`. Thus, the transformed problem is EAC.  $\square$

### 4.2.3 Enforcing Element Hyper-arc Consistency

The procedure for enforcing EHAC is given in Algorithm 4.3. The procedure is similar to the algorithm for enforcing EAC. All possible pairs of set variable and set element are inserted to  $Q$ . Each time a pair  $(S_i, a)$  is picked out from  $Q$  and ternary supports are forced for each existence state of set element  $a$  in set variable  $S_j$  and  $S_k$ . Supports are found by sending cost from the ternary element cost function to unary cost function in `FindTernarySupports()`. As the cost of unary cost function is changed, `FindUnarySupports()` is called to maintain ENC. Lastly, each set element in all set variables is checked in `PruneVar()` to remove any inconsistent set element.

The procedure `FindTernarySupports()` has complexity  $\mathcal{O}(1)$  as each set element has a maximum of two existence states and the procedure only handles ternary supports. Given a WCSP of  $n$  set variables, each of which has a maximum of  $e$  set elements. EHAC() has complexity  $\mathcal{O}(ne + (ne)(n^2 + ne)) = \mathcal{O}(n^2e(n + e))$ .

**Theorem 4.5** Given a WCSP  $P$ , Algorithm 4.3 transforms  $P$  to  $P'$  such that

1.  $P'$  is equivalent to  $P$ , and
2.  $P'$  is EHAC.

**Proof** The procedure given in Algorithm 4.3 only involves basic operations on the costs. In `FindTernarySupports()`, for each existence state  $\beta \in E(S_j, a)$ ,



---

**Algorithm 4.3:** Enforcing element hyper-arc consistency

---

```

1 Procedure FindTernarySupports( $S_i, S_j, a$ )
2 begin
3   for  $\beta \in E(S_j, a)$  do
4      $c := \min_{\alpha \in E(S_i, a), \gamma \in E(S_k, a)} (\varphi_{(i,j,k)/a}(\alpha, \beta, \gamma))$ 
5      $\varphi_{(j)/a}(\beta) := \varphi_{(j)/a}(\beta) \oplus c$ 
6     for  $\alpha \in E(S_i, a), \gamma \in E(S_k, a)$  do
7        $\varphi_{(i,j,k)/a}(\alpha, \beta, \gamma) := \varphi_{(i,j,k)/a}(\alpha, \beta, \gamma) \ominus c$ 
8   FindUnarySupports( $S_j, a$ )
9   for  $\gamma \in E(S_k, a)$  do
10     $c := \min_{\alpha \in E(S_i, a), \beta \in E(S_j, a)} (\varphi_{(i,j,k)/a}(\alpha, \beta, \gamma))$ 
11     $\varphi_{(k)/a}(\gamma) := \varphi_{(k)/a}(\gamma) \oplus c$ 
12    for  $\alpha \in E(S_i, a), \beta \in E(S_j, a)$  do
13       $\varphi_{(i,j,k)/a}(\alpha, \beta, \gamma) := \varphi_{(i,j,k)/a}(\alpha, \beta, \gamma) \ominus c$ 
14  FindUnarySupports( $S_k, a$ )
15 end

16 Procedure EHAC( $\mathcal{X}, \mathcal{D}, \mathcal{C}$ )
17 begin
18    $Q := \emptyset$ 
19   for  $S_i \in \mathcal{X}$  do
20     for  $a \in \mathcal{U}_i$  do
21        $Q := Q \cup \{(S_i, a)\}$ 
22   while  $Q \neq \emptyset$  do
23      $(S_i, a) \in Q$ 
24      $Q := Q \setminus \{(S_i, a)\}$ 
25     for  $C_{i,j,k} \in \mathcal{C}$  do
26       FindTernarySupports( $S_i, S_j, S_k, a$ )
27     for  $S_m \in \mathcal{X}$  do
28       for  $a \in \mathcal{U}_k$  do
29         if PruneVar( $S_m, a$ ) then
30            $Q := Q \cup \{(S_m, a)\}$ 
31 end

```

---



the minimum cost of  $\varphi_{(i,j,k)/a}(\alpha, \beta, \gamma)$  for  $\alpha \in E(S_i, a), \gamma \in E(S_k, a)$  is added to  $\varphi(j)/a(\beta)$ . At the same time, the same amount of cost is subtracted from  $\varphi(i, j, k)/a(\alpha, \beta, \gamma)$  for  $\alpha \in E(S_i, a), \gamma \in E(S_k, a)$ . Similar process is applied for each existence state  $\gamma \in E(S_k, a)$ . An equivalent on cost evaluation is preserved.

The cost operations in `FindTernarySupports()` ensure that there is a ternary support for each existence state of a set element for a set variable. Inconsistent existence states are pruned in `PruneVar()`. Thus, the transformed problem is EHAC.  $\square$

#### 4.2.4 Enforcing Weighted Cardinality Consistency

Algorithm 4.4 gives the procedure for enforcing weighted cardinality consistency. In procedure `WCC()`, `ReviseCardinality()` is called for each set variable. First, the cardinality lower and upper bounds are reset so that they are within the interval  $\{|RS(S_i)|, \dots, |PS(S_i)|\}$ . Then, the cardinality bounds are reduced if the values of cardinality on the bounds are inconsistent. Lastly, when the cardinality lower and upper bounds are equal to each other, the cardinality of the set variable is fixed and the cost of cardinality is send to the global lower bound,  $C_\emptyset$ . Suppose  $ub(|S_i|) = lb(|S_i|) = k$ . If  $k = |RS(S_i)|$ , then  $S_i$  is fixed to  $RS(S_i)$ ; otherwise if  $k = |PS(S_i)|$ ,  $S_i$  is fixed to  $PS(S_i)$ .

The complexity of `ReviseCardinality()` is  $\mathcal{O}(e)$  for each set variable with a maximum of  $e$  set elements. When a WCSP has  $n$  set variables, `WCC()` has complexity  $\mathcal{O}(ne)$ .

**Theorem 4.6** Given a WCSP  $P$ , Algorithm 4.4 transforms  $P$  to  $P'$  such that

1.  $P'$  is equivalent to  $P$ , and
2.  $P'$  is WCC.

---

**Algorithm 4.4:** Enforcing weighted cardinality consistency
 

---

```

1 Procedure ReviseCardinality( $S_i$ )
2 begin
3    $lb(|S_i|) := \max(lb(|S_i|), |RS(S_i)|)$ 
4    $ub(|S_i|) := \min(ub(|S_i|), |PS(S_i)|)$ 
5   while  $Cost_{|i|}(lb(|S_i|)) \oplus C_\emptyset = \top$  do
6      $lb(|S_i|) := lb(|S_i|) + 1$ 
7   while  $Cost_{|i|}(ub(|S_i|)) \oplus C_\emptyset = \top$  do
8      $ub(|S_i|) := ub(|S_i|) - 1$ 
9   if  $lb(|S_i|) = ub(|S_i|)$  then
10     $C_\emptyset := C_\emptyset \oplus Cost_{|i|}(lb(|S_i|))$ 
11     $Cost_{|i|}(lb(|S_i|)) := 0$ 
12    if  $lb(|S_i|) = |RS(S_i)|$  then
13       $PS(S_i) := RS(S_i)$ 
14    if  $ub(|S_i|) = |PS(S_i)|$  then
15       $RS(S_i) := PS(S_i)$ 
16 end

17 Procedure WCC( $\mathcal{X}, \mathcal{D}, \mathcal{C}$ )
18 begin
19   for  $S_i \in X$  do
20     ReviseCardinality( $S_i$ )
21 end

```

---

**Proof** The procedure given in Algorithm 4.4 removes any inconsistency values of cardinality in `FindTernarySupports()`. Both  $lb(|S_i|)$  and  $ub(|S_i|)$  are first revised so that they are within  $\{|RS(S_i)|, \dots, |PS(S_i)|\}$ . The pruning of values of cardinality is done by sequential checking starting from the two bounds. Only inconsistency values of cardinality are removed. Cost is only transferred from the cardinality constraint to the global lower bound when the cardinality is fixed. Thus the transformed problem  $P'$  is equivalent to  $P$  and is WCC.  $\square$

#### 4.2.5 Enforcing Weighted Set Bounds Consistency

---

**Algorithm 4.5:** Enforcing weighted set bounds consistency (Part 1)

---

```

1 Procedure ReviseCardinalityForWSBC( $S_i, Q$ )
2 begin
3    $lb(|S_i|) := \max(lb(|S_i|), |RS(S_i)|)$ 
4    $ub(|S_i|) := \min(ub(|S_i|), |PS(S_i)|)$ 
5   while  $Cost_{|i|}(lb(|S_i|)) \oplus C_\emptyset = \top$  do
6      $lb(|S_i|) := lb(|S_i|) + 1$ 
7   while  $Cost_{|i|}(ub(|S_i|)) \oplus C_\emptyset = \top$  do
8      $ub(|S_i|) := ub(|S_i|) - 1$ 
9   if  $lb(|S_i|) = ub(|S_i|)$  then
10     $C_\emptyset := C_\emptyset \oplus Cost_{|i|}(lb|S_i|)$ 
11     $Cost_{|i|}(lb|S_i|) = 0$ 
12    if  $lb(|S_i|) = |RS(S_i)|$  then
13      for  $a \in PS(S_i) \setminus RS(S_i)$  do
14         $Q := Q \cup \{(S_i, a)\}$ 
15         $PS(S_i) := RS(S_i)$ 
16    if  $ub(|S_i|) = |PS(S_i)|$  then
17      for  $a \in PS(S_i) \setminus RS(S_i)$  do
18         $Q := Q \cup \{(S_i, a)\}$ 
19         $RS(S_i) := PS(S_i)$ 
20 end

```

---



---

**Algorithm 4.6:** Enforcing weighted set bounds consistency (Part 2)

---

```

1 Procedure WSBC( $\mathcal{X}, \mathcal{D}, \mathcal{C}$ )
2 begin
3    $Q := \emptyset$ 
4   for  $S_i \in \mathcal{X}$  do
5     for  $a \in \mathcal{U}_i$  do
6        $Q := Q \cup \{(S_i, a)\}$ 
7   while  $Q \neq \emptyset$  do
8      $(S_i, a) \in Q$ 
9      $Q := Q \setminus \{(S_i, a)\}$ 
10    FindUnarySupports( $S_i, a$ )
11    for  $C_{i,j} \in \mathcal{C}$  do
12      FindBinarySupports( $S_i, S_j, a$ )
13    for  $C_{i,j,k} \in \mathcal{C}$  do
14      FindTernarySupports( $S_i, S_j, S_k, a$ )
15    for  $S_m \in \mathcal{X}$  do
16      for  $a \in \mathcal{U}_m$  do
17        if PruneVar( $S_m, a$ ) then
18           $Q := Q \cup \{(S_m, a)\}$ 
19          ReviseCardinalityForWSBC( $S_m, Q$ )
20 end

```

---

The procedure for enforcing weighted set bounds consistency is depicted in Algorithms 4.5 and 4.6. The procedure `WSBC()` incorporates algorithms for enforcing ENC, EAC and EHAC by calling functions `FindUnarySupports()`, `FindBinarySupports()` and `FindTernarySupports()` for each pair of set variable and set element in the problem. In each iteration, the global lower bound may be changed after these functions are called. The algorithm scans for all pairs of set variable and set element to prune any inconsistent existence states. While some set domains have been changed which may affect the bounds of the domain. `ReviseCardinalityForWSBC()` is called to ensure the set variable is weighted cardinality consistent. It is a slight modification of the original `ReviseCardinality()`. It inserts pairs of set variable and set element to  $Q$  whenever there are changed in their domains.

The complexity of `ReviseCardinalityForWSBC()` is  $\mathcal{O}(e)$  for a set variable  $S_i$  with a maximum of  $e$  set elements throughout the running of the algorithm. Thus, the whole algorithm, `WSBC()`, has complexity  $\mathcal{O}(ne + ne(1 + n + n^2 + ne + ne)) = \mathcal{O}(n^2e(n + e))$ .

**Theorem 4.7** Given a WCSP  $P$ , Algorithms 4.5 and 4.6 transforms  $P$  to  $P'$  such that

1.  $P'$  is equivalent to  $P$ , and
2.  $P'$  is WSBC.

**Proof** The procedure given in Algorithms 4.5 and 4.6 incorporates the procedures for enforcing ENC, EAC, EHAC and WCC. By Theorem 4.1, WSBC is enforced when a problem is ENC, EAC, EHAC and WCC.  $\square$

## Chapter 5

# Experiments

We modified ToolBar [BHdG<sup>+</sup>04], a generic integer WCSP solver, to handle also set variables and conducted experiments to verify the feasibility of our proposal. The comparison is made among our prototype implementation, the original ToolBar and ILOG Solver 6.0 [ILO03] (for classical cases only). While our implementation and ILOG Solver use set variables in modeling, the problems are transformed to use 0-1 variables for the original ToolBar to solve. In the rest of this chapter, we refer to our implementation as ToolBar-Set, the original ToolBar as ToolBar-01, and ILOG Solver as ILOG.

We experimented on the Steiner Triple System and the Social Golfer Problem, which are well known set CSP benchmarks. We solved for all solutions to make our results independent of search heuristics. Besides solving the problems as classical CSPs, we generated two soften versions for each instance to compare the performance of solving WCSPs with set variables.

The experiments were conducted on a Sun Blade 2500 ( $2 \times 1.6\text{GHz}$  US-IIIi) machine with 2GB memory. We report the runtime in seconds and number of fails in solving the problems. The time limit for solving each instance is 600 seconds. In each table, we use ‘-’ to indicate non-termination within the time limit. The shortest runtime are highlighted in bold for each problem instance.



## 5.1 Modeling Set Variables Using 0-1 Variables

ToolBar [BHdG<sup>+</sup>04] is a generic and efficient WCSP solver for solving integer WCSPs. As ToolBar cannot handle set variables directly, we have to model the problem using 0-1 variables. A 0-1 variable is an integer variable with domain  $\{0,1\}$ .

The modeling approach is straight-forward. Whenever we have a set variable  $S_i$  with  $e$  possible set elements in the original set model, we use  $e$  0-1 variables  $x_{i_1}, \dots, x_{i_e}$  to represent  $S_i$  in a 0-1 model by the following relation :

$$\forall a \in \mathcal{U}_i, a \in S_i \text{ if and only if } x_{i_a} = 1$$

For each set variable  $S_i$  with a unary set constraint  $C_i$ , we have, for each set element  $a \in \mathcal{U}_i$ , a unary element cost function  $\varphi_{(i)/a}$  in set model and a unary constraint  $C_{i_a}$  in 0-1 model such that :

$$\varphi_{(i)/a}(t) = C_{i_a}(1)$$

$$\varphi_{(i)/a}(f) = C_{i_a}(0)$$

For each set variable  $S_i$  with a binary set constraint  $C_{i,j}$ , we have, for each set element  $a \in \mathcal{U}_i$ , a binary element cost function  $\varphi_{(i,j)/a}$  in set model and a binary constraint  $C_{i_a,j_a}$  in 0-1 model such that, for  $\alpha \in \{t, f\}$  :

$$\varphi_{(i,j)/a}(t, \alpha) = C_{i_a,j_a}(1, \beta)$$

$$\varphi_{(i,j)/a}(f, \alpha) = C_{i_a,j_a}(0, \beta)$$

where  $\beta \in \{0, 1\}$  and  $\alpha = t$  if and only if  $\beta = 1$ .

The transformation is similar for ternary constraints. We transform a cardinality constraint  $C_{|i|}$  for  $S_i$  where  $|\mathcal{U}_i| = e$  in the set model to an  $e$ -ary

constraint  $C_{i_1 \dots i_n}$  such that

$$C_{|u|}(u) = C_{i_1 \dots i_n}(\eta_{i_1}, \dots, \eta_{i_n})$$

where  $a \in u$  if and only if  $\eta_a = 1$ .

## 5.2 Softening the Problems

Steiner Triple System and the Social Golfer Problem are classical problems containing only classical constraints. In order to generate soft versions of the problem, we can impose more restrictions or relax the existing constraints. A constraint becomes more restricted when we add preferences for the values in the variable domains. A problem becomes more relaxed when we reduce the costs in the constraints.

In the following experiments, we have two versions of softened problems : *Restricted* and *Relaxed*. A *Restricted* version of a problem is generated by randomly adding costs from 0 to 9 to the unary constraints. The original problem is transformed such that we have preferences to the values. The solution space is reduced since the constraints are more restrictive. On the other hand, a *Relaxed* version of a problem is generated by randomly replacing costs from 1 to  $\top$  whenever the cost is  $\top$  in the original constraints. This increases the search space as costs for violating a constraint is reduced from  $\top$  to a cost in  $\{1, \dots, \top\}$ . To measure the runtime for these two versions, we generated 10 instances for each problem instance and report the average runtime and average number of fails.

### 5.3 Steiner Triple System

The Steiner Triple System (prob044 in CSPLib [GW99]) of order  $n$  is to find a set of  $n(n-1)/6$  triples of distinct integer elements in  $\{1, \dots, n\}$  such that no two triples have more than one common element. A Steiner triple system of order  $n$  exists when  $n$  modulo 6 equals to 1 or 3 [LR80]. An example solution for  $n = 7$  is :

$$\{\{1, 2, 3\}, \{1, 4, 5\}, \{1, 6, 7\}, \{2, 4, 6\}, \{2, 5, 7\}, \{3, 4, 7\}, \{3, 5, 6\}\}$$

We can model the problem as

- Variables :

- Sets of triples in the problem :

$$S_i, i \in [1, \dots, n(n-1)/6]$$

- Auxiliary variables :

$$A_{ij}, \forall i, j \in [1, \dots, n(n-1)/6] \wedge i < j$$

- Domains :

- $D(S_i) = D(A_{ij}) = [\emptyset, \{1, \dots, n\}]$

- Constraints :

$$\forall i, j \in [1, \dots, n(n-1)/6] \wedge i < j$$

- Each auxiliary variable holds the intersection of each pair of triples :

$$S_i \cap S_j = A_{ij}$$

- Each set contains exactly 3 elements :

$$|S_i| = 3$$

- Any two triples have at most one common element :

$$|A_{ij}| \leq 1$$



	Classical					
<b>n</b>	ILOG		ToolBar-Set		ToolBar-01	
	Time	Fails	Time	Fails	Time	Fails
6	0.10	6195	<b>0.05</b>	6195	1.64	7858
7	31.52	1405878	<b>16.84</b>	1405878	-	-

Table 5.1: Runtime and number of fails for solving classical Steiner Triple System

	Restricted				Relaxed			
<b>n</b>	ToolBar-Set		ToolBar-01		ToolBar-Set		ToolBar-01	
	Time	Fails	Time	Fails	Time	Fails	Time	Fails
6	<b>0.05</b>	6195	1.68	7858	<b>0.21</b>	35910	2.84	13744
7	<b>5.40</b>	524729	263.51	490587	<b>46.17</b>	6619628	-	-

Table 5.2: Runtime and number of fails for solving soft Steiner Triple System

We focus on the problems up to order 7 due to the long solving time for the problem of order 9. The runtime and number of fails of solving the problem for all solutions are listed in Tables 5.1 and 5.2. We can observe that the runtime of our implementation is about two times faster than ILOG Solver. Our implementation is faster than that of the original ToolBar implementation in order of two magnitudes.

## 5.4 Social Golfer Problem

The Social Golfer Problem (prob010 in CSPLib [GW99]) is to schedule  $g$  groups of  $s$  golfers over  $w$  weeks so that no two golfers play in the same group twice. The problem can be characterized by  $g$ - $s$ - $w$ . We denote each player with an integer and use brackets to hold the players in each group. A solution for the instance 3-2-3, which is 3 groups of 2 golfers for 3 weeks, of the problem is as

follows :

Week 1 : (1 2) (3 4) (5 6)

Week 2 : (1 3) (2 5) (4 6)

Week 3 : (1 6) (2 3) (4 5)

We can model the problem as

- Variables :

- The  $i$  group of player in week  $j$  :

$$G_{ij}, i \in \{1, \dots, g\}, j \in \{1, \dots, w\}$$

- Auxiliary variables :

$$A_{ij,kl}, i, k \in \{1, \dots, g\}, j, l \in \{1, \dots, w\}, i \neq k$$

- Domains :

- $D(G_{ij}) = D(A_{ij,kl}) = [\emptyset, \{1, \dots, gs\}]$

- Constraints :

- Each group has size exactly  $s$  :

$$|G_{ij}| = s$$

- Groups in the same week should contain distinct player :

$$G_{ij} \cap G_{ik} = \emptyset, j \neq k$$

- Each auxiliary variable holds the intersection of two groups :

$$G_{ij} \cap G_{kl} = A_{ij,kl}, i \neq k$$

- Any two groups can share one player at most :

$$|A_{ij,kl}| \leq 1$$

We reduce the search space of the problems by pre-assigning the players for the first week. Tables 5.3 and 5.4 show the runtime and number of fails

g-s-w	Classical					
	ILOG		ToolBar-Set		ToolBar-01	
	Time	Fails	Time	Fails	Time	Fails
3-2-4	1.26	18449	<b>0.60</b>	18449	52.94	18450
3-2-5	8.66	70289	<b>4.12</b>	70289	-	-
3-3-3	0.28	6817	<b>0.15</b>	6817	11.27	11166
3-3-4	2.22	32737	<b>1.29</b>	32737	231.83	63006
4-2-3	58.49	483461	<b>24.84</b>	483461	-	-
4-3-2	1.46	36145	<b>0.69</b>	36145	43.12	96762
4-4-2	13.10	285865	<b>6.28</b>	285865	545.97	1408596
5-2-2	1.99	10481	<b>0.82</b>	10481	59.93	13474
6-2-2	142.51	563669	<b>55.60</b>	563669	-	-

Table 5.3: Runtime and number of fails for the classical Social Golfer Problem

g-s-w	Restricted				Relaxed			
	ToolBar-Set		ToolBar-01		ToolBar-Set		ToolBar-01	
	Time	Fails	Time	Fails	Time	Fails	Time	Fails
3-2-4	<b>0.13</b>	8063	9.87	6761	<b>0.98</b>	47709	58.15	19915
3-2-5	<b>1.40</b>	58102	192.84	51893	<b>8.00</b>	349072	-	-
3-3-3	<b>0.06</b>	4444	3.51	4294	<b>0.37</b>	30867	15.58	15037
3-3-4	<b>0.58</b>	27253	91.33	31972	<b>5.77</b>	375779	402.20	130965
4-2-3	<b>0.97</b>	104878	63.51	67827	<b>30.27</b>	976112	-	-
4-3-2	<b>0.05</b>	7698	1.45	4416	<b>0.95</b>	60153	46.63	102534
4-4-2	<b>0.27</b>	45147	12.36	29982	<b>14.23</b>	1162358	-	-
5-2-2	<b>0.05</b>	6729	1.77	4234	<b>0.89</b>	17546	61.95	13821
6-2-2	<b>0.81</b>	121928	47.51	80562	<b>59.05</b>	930691	-	-

Table 5.4: Runtime and number of fails for the soft Social Golfer Problem



of solving the problem. In solving classical instances, our implementation is two times faster than ILOG solver. The original ToolBar implementation has slower runtime. It cannot solve some instances within the time limit. The comparison is consistent in solving soften versions of the problem. Our implementation is two orders of magnitude faster than the original ToolBar implementation. When the search space is increased, the original ToolBar implementation fails to solve even more instances.

## 5.5 Discussions

The experimental results show that the performance of our implementation is comparable with ILOG solver in solving classical instances. The runtime of using ILOG solver is about two times the runtime of using our implementation for all the instances in the two benchmark problems. Since the actual implementation and data structures used in ILOG solver are not disclosed, we cannot provide a firm explanation for this phenomenon. However, the constant ratio of the performance between ILOG solver and our implementation suggests that these two solvers are using the same variable and value ordering heuristics and enforcing the same level of local consistency, which is set bounds consistency, during search. In other words, this also verifies that our proposal is reduced to classical case when the costs in the problem are either 0 or  $\top$ .

On the other hand, the comparison between our implementation and the original ToolBar demonstrates the feasibility and efficiency of our proposal. Since our implementation is designed to handle problems with set variables, the performance of our implementation is better than the original ToolBar, which is an integer WCSP solver. ToolBar has poor performance in solving problems with set variables because (1) it does not have consistency enforcing algorithms specialized for set variables, (2) modeling set variables using 0-1 variables

increases the number of variables to the problem, and (3) the propagation of cardinality constraints is very poor as the cardinality constraints are  $n$ -ary constraints for set variables with a maximum of  $n$  set elements.

## Chapter 6

# Related Work

Two classes of research are most related to work described in this dissertation : local consistencies in WCSPs and approaches to solve classical CSPs with set variables. We first introduce other local consistency notions defined in WCSPs in addition to star node consistency and star arc consistency which are introduced in Chapter 2. Then, different approaches to handle set in classical CSPs are described.

### 6.1 Other Consistency Notions in WCSPs

Two basic local consistency notions, star node consistency and star arc consistency for WCSPs are introduced in Chapter 2. Stronger consistencies are available. They are full directional arc consistency and existential directional arc consistency.

#### 6.1.1 Full Directional Arc Consistency

The definition of star arc consistency is based on simple support. Given a binary constraint  $C_{i,j}$ , a value  $b \in D(x_j)$  is a *simple support* for  $a \in D(x_i)$  when  $C_{i,j}(a, b) = \perp$ . A variable  $x_i$  is arc consistent if every value  $a \in D(x_i)$  has



a simple support in constraint  $C_{i,j}$ . In contrast, given a binary constraint  $C_{i,j}$ , a value  $b \in D(x_j)$  is a *full support* for  $a \in D(x_i)$  when  $C_{i,j}(a, b) \oplus C_j(b) = \perp$ . A variable  $x_i$  is *full arc consistent* (FAC) [Lar02, LS04] if every value  $a \in D(x_i)$  has a full support in constraint  $C_{i,j}$ .

A full support  $b \in D(x_j)$  can be forced by sending the unary cost  $C_j(b)$  to binary costs  $C_{i,j}(\alpha, b)$  for all  $\alpha \in D(x_i)$ . This is a reverse process of sending costs from binary constraints to unary constraints in enforcing star arc consistency. However, when value  $a \in D(x_i)$  has a full support  $b \in D(x_j)$  for binary constraint  $C_{i,j}$ ,  $b \in D(x_j)$  may lose its full support in  $D(x_j)$ . There may be a case that both  $a \in D(x_i)$  and  $b \in D(x_j)$  cannot be full arc consistent at the same time which fails to terminate the FAC maintaining process.

The problem can be circumvented if we only enforce full arc consistency in one direction. When the set of variables  $\mathcal{X}$  is totally ordered by  $>$ , we can have full directional arc consistency (FDAC) [Lar02, LS04]. A variable  $x_i$  is *full directional arc consistent* if every value  $a \in D(x_i)$  has a full support in  $C_{i,j}$  such that  $j > i$ . FDAC does not have the problem as in FAC, but FDAC is a weaker consistency notion than FAC.

### 6.1.2 Existential Directional Arc Consistency

As FDAC is a weaker consistency notion, de Givry et al. [dGHZL05] propose existential arc consistency (EAC) which is a stronger consistency notion. A variable  $x_i$  is *existential arc consistent* if there exists  $a \in D(x_i)$  such that  $C_i(a) = \perp$  and it has a full support in constraint  $C_{i,j}$ . When a variable  $x_i$  is not EAC, for each value  $a \in D(x_i)$  such that  $C_i(a) = \perp$ , then  $\forall b \in D(x_j), C_{i,j}(a, b) \oplus C_j(b) > \perp$ . After enforcing full arc consistency, the variable  $X_i$  becomes node inconsistent and cost is sent from unary constraints to the global lower bound  $C_\emptyset$ . De Givry et al. [dGHZL05] also integrate EAC with

FDAC and defines *existential directional arc consistency* (EDAC). A WCSP is EDAC if it is FDAC and EAC.

## 6.2 Classical CSPs with Set Variables

Different reasoning approaches are introduced to increase the efficiency of solving classical CSPs with set variables. Two important reasoning approaches are bounds reasoning and cardinality reasoning.

### 6.2.1 Bounds Reasoning

A set variable with  $n$  possible set elements has a domain size  $2^n$ . Searching solutions in such large domain size is inefficient. Gervet [Ger97] proposes to specify the set domain by an interval. The domain interval for set variable  $S$  is specified by a greatest lower bound and an least upper bound, which are also known as the required set  $RS(S)$  and the possible set  $PS(S)$  respectively. The search, instead of choosing and assigning a set value  $u \in D(S)$  to the set variable  $S$ , narrows the domain interval by choosing a set element  $a \in PS(S) \setminus RS(S)$  and putting  $a$  in  $RS(S)$  or removing  $a$  from  $PS(S)$ . Each set constraint is associated with a projection function. When enforcing local consistency, the bounds of set domains of all variables in the scope of the constraint are modified accordingly. Set bounds consistency is a local consistency notion on set domains such that the domain of a set variable has minimum size and contains all the consistent values with respect to a constraint.

### 6.2.2 Cardinality Reasoning

By specifying cardinality constraint for a set variable  $S$ , we can restrict the possible values of  $|S|$ . However, other set constraints can also restrict the

possible cardinality of a variable. For example, the set constraint  $S_i \subseteq S_j$  specifies  $S_i$  must be a subset of  $S_j$ . By taking account into the cardinalities, the constraint  $S_i \subseteq S_j$  also implies that  $|S_i| \leq |S_j|$ . Some implementations [AB00, Mül01] of set solvers not only reason on the bounds of set domains, but also perform cardinality reasoning. An additional propagation rule is associated with each set constraint for cardinality propagation in those solvers.



## Chapter 7

# Concluding Remarks

Problems involving set variables are common. Set constraint solving techniques are well studied in classical CSPs. The integer WCSP framework can handle soft problems efficiently on the integer domain. However, the current definitions for local consistency is impractical to process set variables in WCSPs. We have proposed our definition of set variables with some local consistency notions. In the following, we conclude the thesis by summarizing our contributions and giving possible directions for future research.

### 7.1 Contributions

First, we give a formal definition of set variables and set constraints in WCSPs. The domain of a set variable in WCSPs is specified as a set interval with the required set and the possible set as the bounds of the interval. Any set that falls within the bounds belongs to the set domain. A set constraint is a cost function which maps a tuple of set values for the corresponding set variables in the scope to a cost. If we express set constraints as cost tables as in the integer domain, the space complexity, which is exponential to the possible number of set elements, is high. We have proposed a compact representation scheme by specifying costs at the element level via element cost functions, which assign

costs according to the existence states of set elements. This greatly reduces the complexity of constraint specification. The cost for tuple with respect to a set constraint can be computed by summing up all the costs from element cost functions. Using this scheme, we can specify set constraints involving the common operators and relations.

Second, enforcing node and arc consistencies on set variables is impractical as in classical CSPs due to the large domain size. We have generalized the classical set bounds notion for WCSPs. Instead of direct reasoning on the bounds of set domains, we enforce local consistencies with element cost functions. We introduce consistency notions at the element level : namely, element node consistency, element arc consistency, and element hyper-arc consistency. We also introduce weighted cardinality consistency notion for cardinality constraints. We show that weighted set bounds consistency with respect to a constraint can be enforced by maintaining the element level consistencies or weighted cardinality consistency accordingly.

Third, we have designed consistency algorithms for enforcing element node, element arc and element hyper-arc consistencies as well as weighted set bounds consistency in WCSPs with set variables. Complexity results and proof of correctness of these algorithms are also given. In order to verify the feasibility and efficiency of our proposal, we incorporate our algorithms into ToolBar [BHdG<sup>+</sup>04], a generic WCSP solver. Experiments confirm that our implementation is two times faster than ILOG in solving most classical set problems and two orders of magnitude faster than the original ToolBar in solving both classical and soft set problems.

## 7.2 Future Work

We have introduced set variables, set constraints and consistency notions for set variables to WCSPs in our work. Set-based WCSPs open up possibilities for future research.

First, our proposal enables bounds reasoning on the variable domains with respect to different set constraints. Integrating cardinality reasoning to the solvers for classical CSPs can increase the performance of searching [MM97, AB00]. Cardinality projection functions can be derived by studying the set constraints. However, the property of a set constraint in WCSP depends on the cost distribution which can differ from one constraint to another. It would be worth investigating the way to extract the information of cardinality restriction from the cost distribution to increase the efficiency of solving WCSPs with set variables.

Second, the local consistency notions which we adopt for element cost functions are modified from the basic node and arc consistencies for WCSPs on the integer domain. It would be interesting to study the benefit of stronger consistency notions at the element level, such as ones based on full directional arc consistency and existential arc consistency.

Third, Hawkins, Lagoon and Stuckey [HLS05] show how set variables in classical CSPs can be represented by reduced ordered binary decision diagrams (ROBDDs), and give efficient algorithm to enforce domain consistency. It will be interesting to study if the same principle can be extended for WCSPs.

Forth, variable and value orderings can have great impact on search efficiency. In our work, we use the basic variable and value ordering. The variables are in lexicographical order while, at branching, we try to put a set element of variable  $S$  from  $PS(S) \setminus RS(S)$  to  $RS(S)$  before removing it from  $PS(S)$ . Since set constraints in WCSPs are expressed in terms of costs, when we select



variable or set element for branching, we can study how costs can help to give better variable and value ordering heuristics.

# List of Symbols

$\top$	The maximum cost, page 16
$\perp$	The minimum cost, page 16
$\{l, \dots, u\}$	A range of integers from $l$ to $u$ , page 7
$[L, U]$	A range of sets from $L$ to $U$ , page 24
$\varphi_{(i_1, \dots, i_k)}/a$	An element cost function involving set variables $S_{i_1}, \dots, S_{i_k}$ on the set element $a$ , page 33
$a \oplus b$	The binary operation which combines costs $a$ and $b$ , page 16
$a \ominus b$	The binary operation which subtracts cost $b$ from cost $a$ , page 19
$\mathcal{C}$	A finite set of constraints, page 6
$C_\emptyset$	A zero-arity constraint, page 16
$C_{i_1, \dots, i_k}$	A constraint involving variables $x_{i_1}, \dots, x_{i_k}$ , page 6
$Card$	A function mapping a set value to its cardinality, page 37

$\mathcal{D}$	A finite set of variable domains, page 6
$D(x_i)$	The set of possible values of variable $x_i$ , page 6
$D_0(x_i)$	The initial domain of variable $x_i$ , page 6
$dom_S(C)$	All the values in the domain of set variable $S$ that satisfy the constraint $C$ , page 27
$E(S, a)$	The set of possible existence states of set element $a$ for set variable $S$ , page 31
$\mathcal{P}$	A constraint satisfaction problem, page 6
$PS(S)$	The possible set of set variable $S$ , page 24
$RS(S)$	The required set of set variable $S$ , page 24
$S(k)$	A valuation structure in WCSP with $\top = k$ , page 16
$S_i$	A set variable in a problem, page 23
$t$	A tuple containing the assignments for a set of variables, page 6
$\mathcal{U}$	The universal set of a problem, page 31
$\mathcal{U}_i$	The universal set of set variable $S_i$ , page 30
$\mathcal{V}(t)$	The cost of a tuple $t$ , page 16



$var(C_{i_1, \dots, i_k})$  The set of variables appearing in the constraint  $C_{i_1, \dots, i_k}$ , page 7

$var(t)$  The set of variables appearing in the tuple  $t$ , page 7

$wdom_S(C)$  All the values in the domain of set variable  $S$  that are consistent with constraint  $C$ , page 46

$\mathcal{X}$  A finite set of variables, page 6

$x_i$  A variable in a problem, page 6

$x_i \mapsto a$  An assignment which assigns variable  $x_i$  with value  $a$ , page 6

- [Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Workshop of Doctoral Students (WDS'99)*, volume Part IV, pages 505–564, Prague, June 1999. MatFyzVestn [full].
- [Bes94] Christian Bessière. Arc consistency and arc consistency for constraint artificial intelligence. *Artificial Intelligence*, 65(1):119–139, 1994. [full].
- [BFH09] Christian Bessière, Eugene C. Freuder, and Jean-Charles Lange. Using constraint meta-problem solving for constraint consistency computation. *Artificial Intelligence*, 161(1):125–143, 2009. [full].
- [BHG09] S. Bourvet, F. Heuss, S. de Gooijer, J. Lemaire, J. Mairiaux, and T. Schiex. ToolBar: constraint-based problem solving. *WCSP* (<http://www.amn.fr/ics/T/ctaweb/?id=114>), 2009. [full]. [100, 73].

# Bibliography

- [AB00] Francisco Azevedo and Pedro Barahona. Modelling digital circuits problems with set constraints. In *Computational Logic*, pages 414–428, 2000. [71, 74]
- [Apt03] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. [9]
- [Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Week of Doctoral Students (WDS99)*, volume Part IV, pages 555–564, Prague, June 1999. MatFyzPress. [10]
- [Bes94] Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994. [14]
- [BFR99] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999. [14]
- [BHdG<sup>+</sup>04] S. Bouveret, F. Heras, S. de Givry, J. Larrosa, M. Sanchez, and T. Schiex. ToolBar: a state-of-the-art platform for WCSP. <http://www.inra.fr/bia/T/degivry/ToolBar.pdf>, 2004. [3, 4, 59, 60, 73]

- [BP81] C.A. Brown and P.W. Purdom Jr. How to search efficiently. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 588–594, 1981. [8]
- [BR75] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Communication of ACM*, 18(11):651–656, 1975. [9]
- [BR01] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 309–315, 2001. [14]
- [BRYZ05] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005. [14]
- [Cle87] John G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987. [32]
- [CS04] Martin C. Cooper and Thomas Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199–227, 2004. [19]
- [dGHZL05] Simon de Givry, Federico Heras, Matthias Zytnicki, and Javier Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted csps. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, IJCAI 2005*, pages 84–89, 2005. [19, 69]
- [DP87] D. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987. [8]



- [Gas77] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, page 457, 1977. [8]
- [GB65] S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, 1965. [8]
- [Ger97] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997. [1, 2, 3, 27, 29, 32, 70]
- [GW99] I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from <http://csplib.cs.strath.ac.uk/>. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99). [62, 63]
- [HE80] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, oct 1980. [9]
- [HLS05] Peter Hawkins, Vitaly Lagoon, and Peter J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24:109–156, 2005. [3, 74]
- [ILO03] ILOG. *ILOG Solver 6.0 Reference Manual*, 2003. [3, 4, 59]
- [Lar02] Javier Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 48–53, Edmonton, Canada, 2002. [3, 16, 19, 20, 69]

- [LR80] C.C. Lindner and A. Rosa. Topics on steiner systems. *Annals of Discrete Mathematics*, 7, 1980. [62]
- [LS03] Javier Larrosa and Thomas Schiex. In the quest of the best form of local consistency for weighted csp. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI 2003*, pages 239–244, 2003. [3, 19, 21]
- [LS04] Javier Larrosa and Thomas Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004. [18, 19, 21, 69]
- [LS06] J.H.M. Lee and C.F.K. Siu. Weighted constraint satisfaction with set variables. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, 2006. [3]
- [Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. [1, 10, 11, 13]
- [MH86] Roger Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986. [14]
- [MM97] Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, 1997. [3, 74]
- [Mül01] Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001. [71]

- [Nad89] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989. [8]
- [Per92] Mark Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53(2-3):329–342, 1992. [14]
- [Pur83] Paul Walton Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21(1-2):117–133, 1983. [9]
- [SFV95] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 1995*, pages 631–639, 1995. [14]
- [SS87] H.S. Stone and J.M. Stone. Efficient search techniques — An empirical study of the  $N$ -queens problem. *IBM Journal of Research and Development*, 31:464–474, 1987. [9]
- [ZM88] Ramin Zabih and David McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 155–160, St. Paul, Minnesota, August 21-26 1988. AAAI Press / The MIT Press. [9]
- [ZY01] Y. Zhang and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 316–321, 2001. [14]





CUHK Libraries



004359209