Quantified Weighted Constraint Satisfaction Problems

MAK, Wai Keung Terrence

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

September 2011

Abstract

Soft constraints are functions returning costs, and are essential in modeling over-constrained and optimization problems. Weighted constraint satisfaction is a soft constraint framework, aiming to find a complete assignment with minimum costs.

Recently, we are interested in exploring constraint frameworks with adversaries. Quantified constraint satisfaction, which associates $\exists$ and $\forall$ quantifiers with variables, is one of these frameworks.

We are interested in tackling soft constrained problems with adversarial conditions. Aiming at generalizing the weighted and quantified constraint satisfaction frameworks, a *Quantified Weighted Constraint Satisfaction Problem* (QWCSP) consists of a set of finite domain variables, a set of soft constraints, and a $\min$ or $\max$ quantifier associated with each of these variables. We formally define QWCSP, and give examples to show how we compute the costs we desire.

We give a complete solver based on alpha-beta pruning, followed by discussions on the general pruning conditions allowing us to further prune the search space. Node and arc consistency notions satisfying these conditions are introduced. Furthermore, we suggest two value ordering heuristics to increase efficiency. These notions and heuristics exploit the semantics of the quantifiers.

QWCSPs are useful special cases of QCOP/QCOP+, and can be solved as a QCOP/QCOP+. Restricting our attention to only QWCSPs, we show empirically that our proposed solving techniques can better exploit problem characteristics than those developed for QCOP/QCOP+. Experimental results confirm the feasibility and efficiency of our proposals.

摘要

軟約束是一種輸出費用的函數，是用來模擬過約束問題以及約束優化問題必不可少的工具。加權約束滿足是軟約束框架下的問題。它的目標是找到一個最小費用的完整解。

最近，我們有興趣探討可以模擬有對抗條件的約束框架問題。而量化約束問題，就是其中一種問題。它用存在量化($\exists$)以及全稱量化($\forall$)來量化變數。

我們有興趣解決有對抗條件的軟約束問題。歸納了加權約束滿足問題以及量化約束滿足問題，我們提出量化加權約束滿足問題(QWCSP)。它是由一組有限的變數，一組軟約束，以及用最少量化(min) 或最大量化(max) 來量化變數的問題。我們給了QWCSP 的正式定義，並給予例子來說明如何計算最終的費用。

我們給一個以$\alpha - \beta$剪枝($\alpha - \beta$ pruning)為基礎的完整解算器，接著討論了一般用來減少搜索空間的條件。我們也介紹了滿足這些條件的節點相容(Node Consistency)和弧相容(Arc Consistency)。此外，我們還提出了兩種提高效率的變數數值排序策略法。這些相容概念和排序策略法是利用量化的語義來制定。

量化加權約束滿足問題是有用的特殊QCOP / QCOP+，並可以以QCOP / QCOP+來解決。把注意力集中在量化加權約束滿足問題，我們證明由我們提出的解決技術比QCOP / QCOP+ 的解決方法可以更好地利用問題的特徵。實驗結果證實了建議的可行性以及解決技術的效率。

Acknowledgments

I sincerely thank my supervisor Professor Jimmy Lee for bringing me into the area of research, and from there, I started my research journey in constraint satisfaction at 2009. Jimmy is always enthusiastic about his teaching and research, and I enjoy his lessons very much. Throughout his teaching and guidance, I gain knowledge not only in doing research, but also in many other areas/aspects, and I enjoy having discussions with him.

I would like to thank Professor Arnaud Lallouet, Professor Ho Fung Leung, and Professor Lap Chi Lau to be my examiners. They provide valuable comments and constructive suggestions to improve my thesis. Future research directions are also suggested and discussed.

I would like to thank May Woo and Charles Siu for their guidance within these years. They patiently listen to my questions. I also thank Wu Yi, Shum Yu Wai, and Li Jing Ying for our daily discussions in our office. “Extreme proof-reading” methods will not be established without them.

Lastly, I would like to give my best wishes to my family for their endless supports throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis reports a framework *Quantified Weighted Constraint Satisfaction Problems* (QWCSPs), which aims to tackle constraint optimization problems with adversaries controlling part of the variables. The proposed framework is based on both Weighted Constraint Satisfaction Problems and Quantified Constraint Satisfaction Problems. This chapter gives a brief introduction on the classical constraint satisfaction framework, weighted constraint satisfaction framework, and followed by the quantified constraint satisfaction framework. We will outline our motivations and goals, and then give an overview of the thesis.

1.1 Constraint Satisfaction Problems

Many combinatorial decision problems and optimization problems can be modeled as Constraint Satisfaction Problems (CSPs). According to the definitions by Mackworth [26], we can define CSPs 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.*"

The basic solving algorithm for CSPs is backtracking tree search [1], which traverses the solution space of CSPs by assigning a value to a variable each time. The goal of the search is to find a set of assignments which assigns values to all variables in the CSP, such that all the constraints are satisfied. If the current set of assignments does not satisfy a constraint, the algorithm assigns another value to the last assigned variable (backtracks), otherwise, it extends the current set of assignments by assigning a value to an unassigned variable. The search is complete and sound, however, the execution time of the algorithm can be large. To improve efficiency, Mackworth introduces consistency notions [26]. These notions extract values which must not participate in any solutions, and hence, we can remove (prune) these values before extending the current set of assignments. If the domain of a variable is empty after prunings, this implies the current set of assignments must violate at least one constraint. The searching algorithm will perform backtracks. Node consistency [26] and arc consistency [26] are common consistency notions.

CSPs, in general, have many solutions. To find the best solution of a CSP, one way is to construct a Constraint Optimization Problem (COP) by adding an objective function to the CSP. The objective function maps each complete assignment to a real number. The goal of a COP is to find a complete assignment which satisfies all the constraints, and maximizes (,or minimizes) the objective function. To solve a COP, we augment the backtracking tree search for CSPs by using the objective function to guide search. After finding a solution, we add a constraint to remaining unexplored problems. The constraint restricts all solutions must be better, according to the objective function, than the current solution.

1.2 Weighted Constraint Satisfaction Problems

We often encounter preferences in industrial optimization problems. The goal for these problems is to find solutions minimizing the degree of violation for these preferences. Suppose we directly translate these preferences into constraints, restricting all solutions must satisfy all preferences. The resulting CSP is usually unsatisfiable, as conflicts usually exist among preferences and no solutions satisfy all the preferences.

Soft constraint frameworks (soft CSPs) extend classical CSPs by allowing soft constraints to return costs for each combination of values. Weighted constraint frameworks (Weighted CSPs/ WCSPs) are soft constraint frameworks which further require costs must be positive bounded integers, and the goal is to find a complete assignment minimizing the summation of costs given by soft constraints. Using Weighted CSPs, we can model preferences by soft constraints. We give higher costs for combinations of values violating higher the degree of violation for the preference.

To solve a weighted CSPs, we apply the Branch and Bound algorithm similar to COPs. To further increase efficiency, we define consistency notions which extract costs information and pruning opportunities for soft constraints. Example consistency notions include NC* [23], AC* [23], FDAC* [22], EDAC* [18], and OSAC [16]. To extract costs from soft constraints, two important notions: projections and extensions for enforcing consistency algorithms are used. Projection is an operation which extract costs from higher arity constraints to lower arity constraints, and extension is an operation which extract costs from lower arity constraints to higher arity constraints. Both operations preserve problem equivalence, and allow us to redistribute costs among constraints.

1.3 Quantified Constraint Satisfaction Problems

In planning problems, we may not be able to control all decisions. Environmental changes and unexpected behaviours from adversaries may arise in planning problems and affecting decisions. We classify decisions which are not controlled by us as uncontrollable decisions. For planning problems, we sometimes aim to find a feasible plan for the worst case scenarios, or to find a plan for all scenarios. To tackle this class of problems, we formulate Quantified Constraint Satisfaction Problems (QCSPs) by associating $\exists$ quantifiers and $\forall$ quantifiers to variables in classical CSPs. To model planning problems, we use an existential variable ($\exists x_i$) to represent controllable decisions, and a universal variable ($\forall x_i$) to represent uncontrollable decisions. The semantic of a QCSP is to find assignments for an existential variable, where each assignment corresponds to a combination of values for the universal variables preceding the existential variable, such that all the constraints are satisfied. In general, different ordering of variables results in different QCSPs, and the time complexity of QCSPs raises to PSPACE-complete [15].

To solve a QCSP, one way is to modify the backtracking tree search for solving CSPs. In this way, we are tackling QCSPs in a top-down approach, by branching values of preceding variables first. QCSP-Solve [21] is a solver using backtracking tree search. Arc consistencies notions [27] and forward checking algorithms [21] for quantified constraints are used in the solver. In addition, several techniques are also developed for solving QCSPs. These include conflict based-backjumping [21], solution-directed backjumping [3], pure literal rule [21], and symmetry breaking techniques [21]. There is also a solver called BlockSolve [44], which uses a bottom up approach tackling QCSPs.

1.4 Motivation and Goal

The task at hand is that of an optimization problem with adversarial conditions. As an example, we begin with a generalized graph coloring problem in which numbers are used instead of colors. In addition, the graph is numbered by two players. The nodes are partitioned into two sets, A and B. Player 1 will number set A first, followed by player 2 numbering set B. The goal of player 1 is to maximize the total difference between numbers of adjacent nodes, while player 2 wishes to minimize the total difference. The aim is to help player 1 devising the best strategy.

The example is optimization in nature, and the adversaries originate from the numbers being placed on nodes by player 2. The question can be translated to maximizing total difference for all possible combinations of numbers player 2 can write. One way to solve this problem is by tackling many Constraint Optimization Problems [1] or Weighted CSPs [23], where each of them maximizes the total difference conditioned on a specific combination of numbers given by player 2. Solving these sub-problems independently, however, defies opportunities for exploiting global problem structure and characteristics, and reuse of computation. Another way is to model the problem as a QCSP [12] by finding whether there exists combinations of numbers for player 1 for all number placements by player 2 such that the total difference is less than a cost k. Trying different values of k progressively in separate QCSPs falls short in utilizing the objective function to guide search globally. We propose to combine the best of both worlds.

Weighted CSPs are minimization in nature. We introduce the $\max$ quantifier to further allow min-max operations on constraint costs in Weighted CSPs. A *Quantified Weighted Constraint Satisfaction Problem* (QWCSP) consists of an ordered sequence of finite domain variables, a set of soft constraints, and a $\min$ or $\max$ quantifier associated with each variable. Note that the existential ($\exists$) and universal ($\forall$) quantifiers in QCSPs can be expressed using $\min$ and $\max$ respectively. WCSPs and QCSPs are thus also special cases of QWCSPs. We define solutions of a QWCSP, and show how branch-and-bound search with alpha-beta pruning can be applied to solve a QWCSP. We introduce node and arc consistency notions and two value ordering heuristics. In addition, general pruning conditions of alpha-beta search are defined and discussed. The new framework aims to answer such interesting questions as "What is the best plan I can choose to minimize all the possible penalties for the worst case scenario?" QWCSPs can be modeled as QCOPs [5], which are a more general framework. We give a construction method followed by an example. We perform experimental evaluations, comparing our proposed solving techniques and those for QCOPs, on three benchmarks to show the efficiency and feasibility of our framework.

1.5 Outline of the Thesis

We now outline of the remaining parts of the thesis.

Chapter 2 provides basic background information for the thesis. Definitions and semantics for CSPs, WCSPs, and QCSPs will be given. We will explain solving algorithms together with consistency notions for these frameworks. Examples will be provided throughout the chapter.

Chapter 3 gives formal definitions for our framework. We will explain the semantics, following by an example. Relationships with QCSPs and WCSPs will be highlighted.

Chapter 4 describes the complete solver for our framework, starting by discussing the basic alpha-beta prunings. Sufficient conditions allowing consistency algorithms to utilize alpha-beta prunings will be discussed. We will then show our node consistency and arc consistency notions, followed by their respective enforcing algorithms. Theorems for the correctness of these notions will be provided, and time complexity for enforcing algorithms will be given.

Chapter 5 discusses performance evaluations on our frameworks and solving techniques. To show the effectiveness of our approach, we compare our work with one of our related frameworks, QCOPs [5]. We will show how to construct QCOPs from QWCSPs, and conjecture that our framework can achieve more prunings by an example. We also give emperical results on three benchmarks, and compare our solver against the solver for solving QCOPs. Results for using different value ordering heuristics are hightlighted.

We conclude our thesis in Chapter 6. We summarize our contributions, and highlight related frameworks aiming to tackle optimization problems with adversaries. Comparisons between our framework and these frameworks will be drawn. Future works, limitations, and potential enhancements for our framework will also be discussed.

Chapter 2

Background

This chapter provides background information for the rest of the thesis. The framework we proposed, Quantified Weighted Constraint Satisfaction Problems (QWCSPs), combines Quantified Constraint Satisfaction Problems (QCSPs) and Weighted Constraint Satisfaction Problems (WCSPs). Both frameworks extend the classical Constraint Satisfaction Problems (CSPs). CSP is a general framework used to model and solve combinatorial problems. WCSP extends CSP by allowing soft constraints, while QCSP extends CSP by allowing quantifiers associated with each variables. We will introduce each of these frameworks, starting from classical CSPs. Definitions, semantics, and examples will be given for each framework. Algorithms and solving techniques will be presented afterwards.

2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* [1] is a triple $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ is a finite set of *variables*, $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ is a set of finite *domains* of possible values for each variable, and $\mathcal{C}$ is a set of *constraints*. Each constraint $C[S] \in \mathcal{C}$ act on a subset of variables $S \subseteq \mathcal{X}$, restricting the allowed combinations of values the subset of variables S can take. We name S as the *scope* of constraint $C[S]$. The *arity* of the constraint $C[S]$ is defined as the number of variables in S, i.e. $|S|$. A constraint is *unary* if its arity is 1, is *binary* if its arity is 2, and is *n-ary* if its arity is n. To simplify notations, we denote a unary constraint $C[\{x_i\}]$ by C_i, and a binary constraint $C[\{x_i, x_j\}]$ by C_{ij}. We write $x_i = v_i$ as an *assignment* assigning value $v_i \in D_i$ to variable x_i, i.e. variable x_i taking value v_i. Let $l = \{x_{i_1} = v_{i_1}, x_{i_2} = v_{i_2}, \ldots, x_{i_m} = v_{i_m}\}$ be a set of assignments. We abuse terminology by saying the *scope* of l ($scope(l)$) to be the set of variables involved in l, i.e. $scope(l) = \{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\}$. We sometimes denote l by a tuple $(v_{i_1}, v_{i_2}, \ldots, v_{i_m})$ if the associated variable for each coordinate in the tuple is clear. A *complete assignment* is an l where $scope(l) = \mathcal{X}$. A *partial assignment* $l[S] = \{x_i = v_i | x_i = v_i \in l \wedge x_i \in S\}$ is a projection of l onto a subset of variables S, where $S \subseteq \mathcal{X}$. A set of assignment l *satisfy* a constraint $C[S]$, where $S \subseteq scope(l)$, if $l[S]$ specifies an allowed combination of values the subset of variables S can take. A *solution* of $\mathcal{P}$ is a complete assignment such that all constraints are satisfied. Example 1 shows a CSP instance. Solutions of a CSP are not necessarily unique.

Example 1. *Given a CSP $\mathcal{P}$ with the set of variables $\mathcal{X} = \{x_1, x_2, x_3\}$, domains $D_1 = [0..4]$, $D_2 = [2..10]$, and $D_3 = [13..20]$, and the set of constraints $\mathcal{C} = \{x_3 \neq 20, x_1+x_2 > 4, x_1+x_2 = x_3\}$. The CSP $\mathcal{P}$ has 3 variables: x_1, x_2, and x_3, and each variable x_i associate with the domain D_i. There are three constraints: one unary, one binary, and one 3-ary constraint, for the CSP $\mathcal{P}$. The unary constraint $x_3 \neq 20$ restricts values of x_3 must not equal to 20. The binary constraint $x_1 + x_2 > 4$ restricts values of x_1 adding to values of x_2 larger than 4. The 3-ary constraint $x_1 + x_2 = x_3$ restricts values of x_1 adding to values of x_2 equal to values of x_3. The set of assignments $\{x_1 = 3, x_2 = 10, x_3 = 13\}$ and $\{x_1 = 4, x_2 = 10, x_3 = 14\}$ are two of the feasible solutions for the above CSP.*

In general, solving CSPs on finite domain is NP-complete, while solving some specific instances are proven to be tractable [33]. We use the famous 4-queens problem (an instance of n-queens problem) to show how we use CSPs to model and solve combinatorial problems.

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

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

Figure 2.1: Solutions for the 4-queens problem

Example 2. *The n-queens problem is to place n queens on an $n \times n$ chess board in a way such that no two queens can be placed on the same row, same column, or same diagonal. For a 4-queens problem, we will use a CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ with 4 variables $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$, each associates with an integer domain [1..4]. A variable x_i represents the row position of the queen in the i^{th} column. The model implicitly enforces no two queens on the same column. We are left to give constraints in $\mathcal{C}$ to enforce no two queens in the same row/diagonal. Constraints $x_i \neq x_j, \forall 1 \leq i \leq j \leq 4$ are used to enforce no two queens in the same row, and constraints $|x_i - x_j| \neq j - i, \forall 1 \leq i \leq j \leq 4$ are used to enforce no two queens in the same diagonal. There are two solutions, $\{x_1 = 2, x_2 = 4, x_3 = 1, x_4 = 3\}$ and $\{x_1 = 3, x_2 = 1, x_3 = 4, x_4 = 2\}$, for the 4-queens problems. Figure 2.1 shows the graphical representation for these two solutions.*

2.1.1 Backtracking Tree Search

A common technique tackling CSPs is to apply the Backtracking Tree Search [1], which traverses the solution space of a CSP in a depth-first left-to-right mannar. The search guarantees to find all solutions of a CSP, and able to prove no solutions exist. Figure 2.2 shows the Backtracking Tree Search.

We use the notation `P` in the function `backtrackingSearch(P)` to denote the input CSP $\mathcal{P}$. The function `getUnassignedVar(P)` is used to return an unassigned variable of $\mathcal{P}$ (symbol `bot` if not exists), and the function `cons(P)` is

```
1 function backtrackingSearch(P):
2   if getUnassignedVar(P) == bot: return (cons(P))? true: false
3   else:
4     xi = getUnassignedVar(P)
5     for v in Di:
6       Ps = P[xi = v]
7       if cons(Ps):
8         if backtrackingSearch(Ps): return true
9     return false
```

Figure 2.2: Backtracking Tree Search for CSPs

used to check whether the current set of assignments l satisfy all constraints $C \in \mathcal{C}$, where $scope(C) \subseteq scope(l)$. We use `P[xi = v]` to denote a function returning the subproblem obtained from $\mathcal{P}$ by assigning value $v \in D_i$ to variable x_i. The main goal of the search is to assign values to unassigned variables incrementally (line 6), and checks whether the current set of assignments leads to solutions (line 7).

The search starts by checking whether all variables are assigned (line 2). If the current complete assignment satisfies all constraints, the search returns true. If there are variables not assigned, the search assigns (line 6) a value of an unassigned variable. After an assignment, if the current set of assignments (current partial assignment) cannot satisfy all constraints, the search will not extends the current partial assignment. Instead, it chooses other unassigned values (the loop in line 5 to 8). The aim of the search is to extend the current set of assignments, by recursively calling itself, until a solution is found. The whole searching process requires exponential time as it may eventually checks all combinations of possible values.

Figure 2.3 shows the Backtracking Tree Search for the 4-queens problem. The algorithm uses lexicographic ordering to choose values and variables for assignment. Variable x_1 is chosen before x_4, and value 1 is chosen before value 4. We only show the nodes being visited by the tree search till the first solution $\{x_1 = 2, x_2 = 4, x_3 = 1, x_4 = 3\}$ is found. When a variable of the 4-queens problem are assigned, we can view the position of the respective queen is fixed. We can, therefore, represent tree nodes in the search tree as a combination of queens' fixed positions on the board. Nodes at the i^{th} level represent sub-problems, where variables $x_1, x_2, \ldots, x_{i-1}$ are assigned, and nodes being marked by 'Failed' mean the corresponding sets of assignments must violate at least one constraint.

There is another type of searching algorithms called local search [1]. Algorithms in this thesis are based on Backtracking Tree Search, therefore, we only outline the basic idea of local search. Local search algorithms, in general, generate complete assignments, and check whether solutions are found. If a generated complete assignment is not a solution, the search alter it until a solution is found. Algorithms generating complete assignments and altering them may be done deterministically depending on problem structure, i.e. not purely in a random way. Two well known methods, repair method and hill climbing method, are commonly used to alter complete assignments which are not solutions. For local searching methods, there are chances for them to trap in local optima. Most methods equip with heuristics or perform restarts to avoid this situation. Similarly, these heuristics may be designed based on problem structures. Adaptive search [43] is one of the local search.

2.1.2 Local Consistencies for solving CSPs

In Figure 2.3, the search space is huge. If the search can perform reasonings by considering properties of CSPs, the search may not need to traverse all of the tree nodes, and the performance increases. Local consistencies are notions identifying properties of CSPs. We then further devise enforcing/filtering algorithms to maintain these properties. By maintaining these properties, the enforcing algorithms will prune values of variables which cannot lead to solutions, and as a result, transform a harder CSP to a easier one and the search space is smaller. Before introducing consistency notions, we give conditions for these enforcing algorithms. These conditions ensure the correctness of enforcing algorithms.

Definition 2.1. *Given two CSPs $\mathcal{P}_1(\mathcal{X}, \mathcal{D}_1, \mathcal{C}_1)$ and $\mathcal{P}_2(\mathcal{X}, \mathcal{D}_2, \mathcal{C}_2)$. The CSP $\mathcal{P}_1$ is equivalent [1] to another CSP $\mathcal{P}_2$ if they have the same set of solutions.*

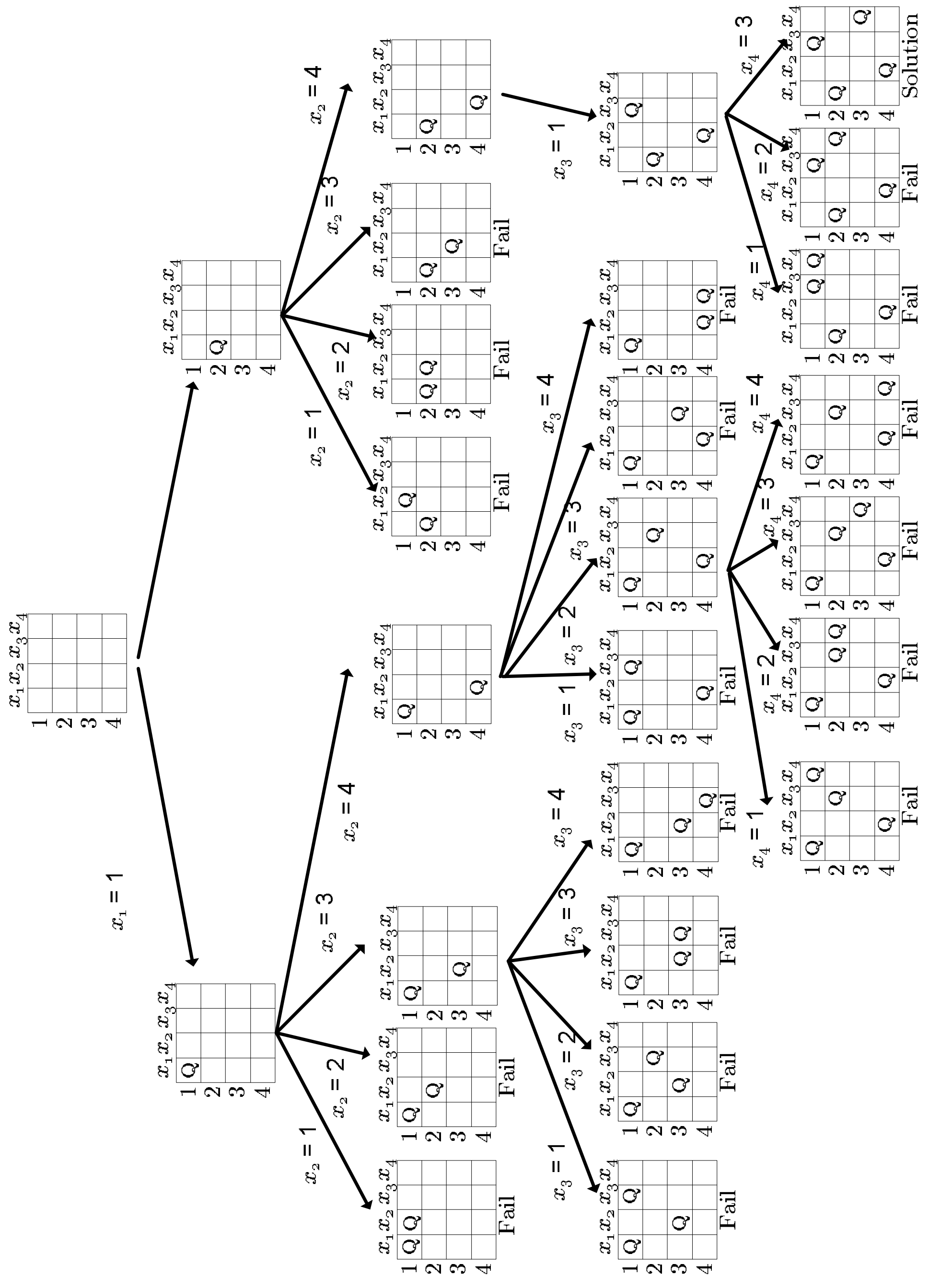

Figure 2.3: Backtracking Tree Search for 4-queens problem

Definition 2.2. *Given a local consistency α. An α-enforcing algorithm is to transform a CSP $\mathcal{P}$ into $\mathcal{P}'$ such that $\mathcal{P}'$ is α and equivalent to P.*

Intuitively, these two definitions prevent enforcing algorithms returning CSPs with different sets of solutions, and ensuring enforcing algorithms return CSPs with the required properties.

We now introduce node consistency for unary constraints, followed by arc consistency for binary constraints. All consistency notions follow the notations in the book 'Principles of Constraint Programming' [1].

Node Consistency (NC)

Definition 2.3. *[26, 1] A value $v_i \in D_i$ is node consistent iff v_i satisfy C_i. A variable x_i is node consistent iff all values in D_i are node consistent. A CSP is node consistent iff all its variables are node consistent.*

Intuitively, we want to prune values which are not node consistent. These values must not participate in any solutions, as they must violate a unary constraint.

Example 3. *Given a CSP $\mathcal{P}$ with the set of variables $\mathcal{X} = \{x_1, x_2, x_3\}$, domains $D_1 = \{1, 3, 4\}$, $D_2 = \{4, 5\}$, and $D_3 = \{13, 17, 18\}$, and the set of constraints $\mathcal{C} = \{x_3 > 14, x_1 + x_2 < 6\}$. Value 13 of variable x_3 in $\mathcal{P}$ is not node consistent. The CSP $\mathcal{P}$, therefore, is not node consistent.*

To devise an enforcing/filtering algorithm for node consistency, one way is to enumerate all unary constraints C_i in a CSP $\mathcal{P}$, and then prune all values $v_i \in D_i$ which do not satisfy C_i. Figure 2.4 [26] shows an enforcing algorithm for NC. The function `cons(Ci,vi)` checks if the value `vi` satisfy constraint `Ci`, and the function `prune(xi,vi)` prune value `vi` of variable `xi`. It is easy to check after applying the algorithm in Figure 2.4, the resulting CSP is node consistent, and is equivalent to the original CSP.

```
function NC(P):
  for xi in [1..n]:
    if Ci exists:
      for vi in Di:
        if cons(Ci,vi) == false:
        prune(xi,vi)
```

Figure 2.4: Node Consistency Enforcing Algorithm

Arc Consistency (AC)

Definition 2.4. *[26, 1] A binary constraint C_{ij} is arc consistent iff: 1) $\forall v_i \in D_i, \exists v_j \in D_j \quad s.t. \quad \{x_i = v_i, x_j = v_j\}$ satisfies C_{ij}, and 2) $\forall v_j \in D_j, \exists v_i \in D_i \ \ s.t. \ \{x_i = v_i, x_j = v_j\}$ satisfies C_{ij}. A CSP $\mathcal{P}$ is arc consistent iff all its binary constraint are arc consistent.*

We also give another definition which is also commonly used to define arc consistency.

Definition 2.5. *A value v_j of variable x_j is a support for value v_i of variable x_i iff the set of assignments $\{x_i = v_i, x_j = v_j\}$ satisfies the constraint C_{ij}. A binary constraint C_{ij} is arc consistent iff every value $v_i \in D_i$ has at least one support $v_j \in D_j$, and every value $v_j \in D_j$ has at least one support $v_i \in D_i$. A CSP $\mathcal{P}$ is arc consistent iff all its binary constraint are arc consistent.*

Intuitively, we want to find values of variables which must violate at least one binary constraint. If a value $v_i \in D_i$ does not have any supports from values $v_j \in D_j$ w.r.t. C_{ij}, then we know the assignment $x_i = v_i$ must not be in any solutions. As seen in Example 3, we have one binary constraint $x_1 + x_2 < 6$ on variable x_1 and x_2. Value 3 and 4 of x_1 do not have any supports from values in D_2, and value 5 of x_2 does not have any supports from values in D_1. Therefore, the constraint $x_1 + x_2 < 6$ is not arc consistent.

Figure 2.5 shows the classical AC enforcing algorithm, `AC-3` [26]. The algorithm maintains a queue `Q` storing all pairs of variables (i, j) where variable x_i does not guarantee to have supports from values of x_j. At each iteration (the while loop

```
1  function AC-3(P):
2    NC(P)
3    Q = {(i,j),(j,i)|Cij in C}
4    while Q != null:
5      (k,l) = pop(Q)
6      if Revise(P,k,l):
7        add(Q,k,l)
8  function Revise(P,k,l):
9    delete = false
10   for vk in Dk:
11     if noSupport(vk,Ckl):
12       prune(xk,vk)
13       delete = true
14   return delete
```

Figure 2.5: Arc Consistency Enforcing Algorithm

in lines 4 to 7), we retrieve a pair of variables (k, l) by using the `pop(Q)` function on the queue `Q`, and invoke the function `Revise(P,k,l)` to find supports (lines 10 to 13) for values of variable x_k. The function `noSupport(vk,Ckl)` checks whether value `vk` of variable x_k has any supports from values in D_l w.r.t. constraint `Ckl`. If there are no supports, the function returns true. The algorithm will then prune value v_k (line 12). If a value v_k of variable x_k is pruned as there are no supports from values of x_l, values of other variables $x_m, m \neq l \wedge m \neq k$ may lose supports w.r.t. constraint C_{km}. There are chances for values of x_m using v_k as supports. We need to recheck by finding supports of x_m. The function `add(Q,k,l)` (line 7) will add pairs of variable (m, k) where $m \neq k \wedge m \neq l \wedge C_{mk} \in \mathcal{C}$ to the queue `Q` to resolve this issue. The time complexity of `AC-3` is $O(ed^3)$, where e is the number of constraints and d is the maximum domain size.

There are various arc consistency algorithms such as: AC-4 [29], AC-5 [34], AC-6 [6], AC-7 [8], AC-2001 [7], AC-3.1 [49], and AC-2001$\backslash$3.1 [9]. Some of these algorithms, for example AC-2001, speed up the run-time for the function `Revise(P,k,l)` by utilizing data structures (hence memory) to store previously found supports.

There are also other types of consistency algorithms (e.g. GAC [30]) for high arity constraints, i.e. constraints covering on more than two variables. As this thesis focuses mainly on unary and binary constraints, we skip detail definitions for these algorithms.

Searching by Maintaining Arc Consistency

When values of a CSP are being pruned, the associated search tree is smaller and the backtracking tree search can traverse faster. Enforcing local consistencies are useful as it may prune values of variables in a CSP and result in a smaller search tree.

One way to combine search and local consistencies is to: 1) apply the NC and AC enforcing algorithms once to obtain a smaller CSP, and 2) use backtracking search to traverse the solution space of the smaller CSP. This method falls short in exploring pruning opportunities during search. When a value of an NC and AC CSP is assigned (during search), the obtained sub-problem is NC, but are not guaranteed to be AC. If we further require enforcing AC in every node of the search tree, we can achieve a smaller search tree. This leads to another method 'Maintaining Arc Consistency (MAC) [39]', aiming to maintain AC during search. We show the algorithm for MAC in Figure 2.6.

The algorithm maintains AC during search by calling the AC enforcing algorithm in line 6 before assigning values. We assume the NC enforcing algorithm is called before calling the MAC algorithm. This allows the AC enforcing algorithm to prune infeasible values for sub-problems obtained during search. If there exists a domain which is empty after pruning values, the AC enforcing algorithm will trigger backtrack (by breaking the for loop). We use the function `P[xi != v]` to prune value `v` of `xi` in this algorithm. The search aims to maintain AC at each tree node, and prevents searching unnecessary assignments. When an assignment $x_i = v$ is explored without finding a solution, the search prunes value v of variable x_i (line 10) during backtrack. As values of other variables may use value v of x_i as support, the search re-runs the AC enforcing algorithm before assigning other values. Figure 2.7 shows the search tree for the 4-queens problem by using the algo-

```
function MAC(P):
  if getUnassignedVar(P) == bot: return (cons(P))? true: false
  else:
    xi = getUnassignedVar(P)
    for v in Di:
      AC-3(P)
      Ps = P[xi = v]
      if cons(Ps):
        if MAC(Ps): return true
      P[xi != v]
    return false
```

Figure 2.6: Searching By Maintaining Arc Consistency

rithm in Figure 2.6. We indicate values removed by the AC enforcing algorithm by a cross ‘X’. We can see the search tree is smaller by comparing with Figure 2.3. For example, in order to prove the assignment $x_1 = 1$ cannot lead to solutions, the improved algorithm only needs to traverse one nodes. For the backtracking algorithm, we need to traverse 17 nodes before concluding failure.

2.1.3 Constraint Optimization Problems

A CSP may have more than one solution, and in some cases, we are interested in finding the best one. Optimization version of CSPs is used to model such scenarios. A *constraint optimization problem* [37] (COP) $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$ is a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, together with an *objective function* f that maps complete assignments to real numbers. An *optimal* solution to a minimization (maximization resp.) COP is a solution l to $\mathcal{P}$ that minimize (maximizes resp.) the value of $f(l)$. COPs is one of the well-known frameworks for optimization.

Example 4. *As an example, we modify Example 2 by adding an objective function f which returns a product multiplying the values of all variables. Therefore, we can write*

$$f(\{x_1 = v_1, ..., x_n = v_n\}) = \prod_{i=1}^{n} v_i$$

The problem now becomes a COP problem. In Example 2, there are two solutions

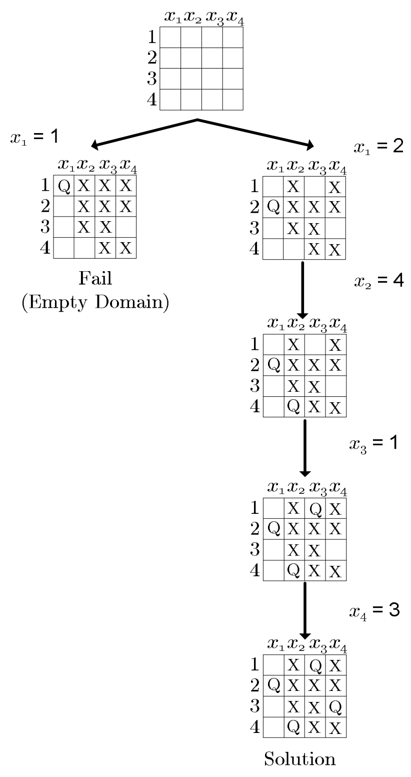

Figure 2.7: Searching by Maintaining AC for 4-queens problem

$\{x_1 = 2, x_2 = 4, x_3 = 1, x_4 = 3\}$ *and* $\{x_1 = 3, x_2 = 1, x_3 = 4, x_4 = 2\}$. *The objective function maps the first solution to 27, and the second solution to 24. If the COP is a minimization (maximization resp.) problem, the second (first resp.) solution is the optimal solution.*

Figure 2.8 shows the branch and bound algorithm [1] for COP (minimization problem). The search can be seen as a modification of the backtracking tree search for CSP with MAC, which is also a depth-first left-to-right search. The algorithm maintains a bound which stores the current best optimal value, and uses a function `addConstraint(obj < bound)` to add a constraint restricting the value returned by the objective function smaller than the bound. Initially, we set the bound to infinity. If a solution is found at the leaf node, the search returns its objective value (line 3). When a solution with better objective value is found, the search: 1) updates the bound (line 12), and 2) adds a constraint to restrict future solutions

```
1  function BranchAndBound(P, bound):
2    if getUnassignedVar(P) == bot:
3      return (cons(P))? obj(P): infinity
4    else:
5      xi = getUnassignedVar(P)
6      for v in Di:
7        AC-3(P)
8        Ps = P[xi = v]
9        if cons(Ps):
10         cost = BranchAndBound(Ps,bound)
11         if cost < bound:
12           bound = cost
13           addConstraint(obj < bound)
14       P[xi != v]
15     return bound
```

Figure 2.8: Backtracking Tree Search for COP

must associate with better objective values (line 13). In this way, the algorithm prevents searching solutions with objective values worse than the current one. After completing the search, the last found solution is the best optimal solution.

Figure 2.9 shows the search tree for the modified 4-queens problem in Example 4. The bound is set to infinity before search. When the first solution $\{x_1 = 2, x_2 = 4, x_3 = 1, x_4 = 3\}$ with objective value 27 is found, the bound is updated to the solution's objective value. A constraint $x_1 \times x_2 \times x_3 \times x_4 < 27$ is added to the remaining unsearched sub-problems. The bound is updated to 24 after the second solution $\{x_1 = 3, x_2 = 1, x_3 = 4, x_4 = 2\}$ is found. As the last found solution is the second solution, the optimal solution is the second one.

2.2 Weighted Constraint Satisfaction Problems

Weighted Constraint Satisfaction Problems (WCSPs) is a framework allowing constraints to return costs. WCSPs extend CSPs by associating costs to tuples of variable assignments.

A *weighted constraint satisfaction problem* [23] (WCSP) is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of *variables* and $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a

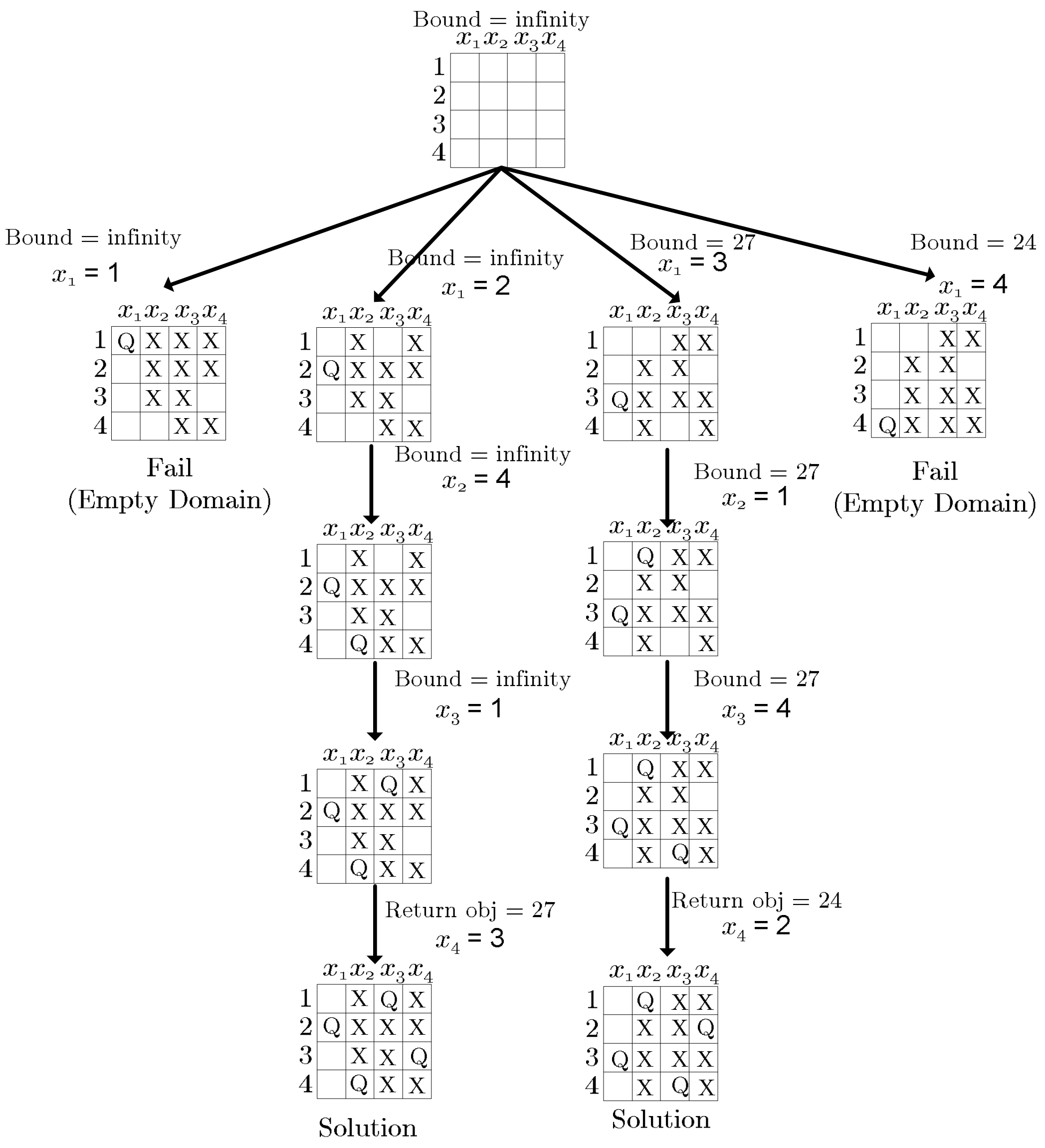

Figure 2.9: Branch and Bound for the modified 4-queens problem

set of *domains* of possible values. We reuse the notions for assignments, scopes of a set of assignments, complete assignments, and partial assignments defined in previous section (Section 2.1). $\mathcal{C}$ is a set of *(soft) constraints*, each $C[S]$ of which represents a function mapping tuples corresponding to assignments on a subset of variables S, to a cost valuation structure $V(k) = ([0..k], \oplus, \leq)$. The structure $V(k)$ contains a set of integers $[0..k]$ with standard integer ordering $\leq$. Addition $\oplus$ is defined by $a \oplus b = \min(k, a + b)$. Subtraction $\ominus$ is defined by $a \ominus b = a - b$ if $a \neq k$ and $k \ominus a = k$ for any $a \geq b$. We reuse the definitions for arities, unary constraints, and binary constraints in Section 2.1. To distinguish constraints between CSPs and WCSPs, we named constraints in CSPs as hard constraints and constraints in WCSPs as soft constraints. We write C_i for the unary constraint on variable x_i, C_{ij} for the binary constraint on variables x_i and x_j, $C_i(d)$ for the cost returned by the unary constraint when d is assigned to x_i, and $C_{ij}(u, v)$ for the cost returned by the binary constraint when u and v is assigned to x_i and x_j respectively. Without loss of generality, we assume the existence of $C_\varnothing$ denoting the lower bound of the minimum cost of the problem. If it is not defined, we assume $C_\varnothing = 0$. The *cost* of a complete assignment l in $\mathcal{X}$ is defined as:

$$cost(l) = C_\varnothing \oplus \bigoplus_{C[s] \in \mathcal{C}} C[s](l[S])$$

A complete assignment l on $\mathcal{X}$ is *feasible* if $cost(l) < k$, and is a *solution* of a WCSP if l has the minimum cost among all feasible tuples.

Example 5. *Given a WCSP $\mathcal{P}$ with the set of variables $\{x_1, x_2, x_3\}$, domains $D_1 = \{a, b, c\}$, $D_2 = \{a, b\}$, and $D_3 = \{a, b\}$, the set of constraints represented in Figure 2.10, and the global upper bound $k = 10$. The problem is to find a complete assignment with minimum costs. Figure 2.10 indicates there are 3 unary constraints C_1, C_2, C_3, and 2 binary constraints $C_{1,2}, C_{2,3}$. For unary constraints, non-zero unary costs are depicted inside a circle and domain values are placed above the circle. For binary constraints, non-zero binary costs are depicted as labels on edges connecting the corresponding pair of values. Only non-zero costs are shown. The* *complete assignment $\{x_1 = a, x_2 = b, x_3 = a\}$ has a cost of 2, and is a solution for the problem $\mathcal{P}$ as it has the minimum costs among all other complete assignments.*

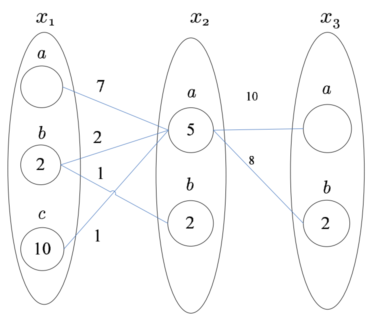

Figure 2.10: WCSP for Example 5

Given a hard constraint C. We can construct a soft constraint C' on the same set of variables. A soft constraint C' returns cost 0 if C is satisfiable on the same set of assignments; otherwise, C' returns cost k.

Theorem 2.6. *A CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ can be transformed by Karp reduction [2] to an equivalent WCSP $\mathcal{P}' = (\mathcal{X}, \mathcal{D}, \mathcal{C}', k)$ with $k = 1$ and $\mathcal{C}'$ is the set of soft constraints constructed from $\mathcal{C}$.*

Proof. We first prove if $\mathcal{P}'$ is unsatisfiable, then $\mathcal{P}$ must be unsatisfiable. The proof for the other case follows afterwards. The equivalent WCSP $\mathcal{P}'$ has the same set of variables and domains. Suppose $\mathcal{P}'$ has no solutions, i.e. there are no feasible complete assignments for $\mathcal{P}'$. This implies there are no complete assignments l s.t. $cost(l) < 1$. We can conclude that for all complete assignments, there exists a soft constraint $C'[S]$ s.t. $C'[S](l[S]) = 1$. Since all soft constraints are constructed from hard constraints C, therefore, all complete assignments must violate at least one hard constraint in $\mathcal{P}$. This implies $\mathcal{P}$ is unsatisfiable. We now prove the other cases. Suppose $\mathcal{P}'$ has a solution l, i.e. $cost(l) < 1$. This implies for all constraint $C'[S] \in \mathcal{C}'$, $C'[S](l[S]) = 0$; otherwise, $cost(l)$ must not less than 1. Since the set of soft constraints C' are constructed from the set of hard constraints C, the complete assignment l must satisfy all hard constraints C. This implies $\mathcal{P}$ is satisfiable. $\square$

2.2.1 Branch and Bound Search (B&B)

To solve WCSPs, we can reuse and modify the Branch and Bound for solving COP. Figure 2.11 shows a high level abstraction of the modified search. The search maintains a bound which stores the current best costs found, and initially, the bound is set to the global upper bound k of the problem.

Similar to the B&B for COP, if all variables are assigned, the search returns the costs of the current complete assignment. This can be achieved by function `cost(P)` in line 2. If there are variables unassigned, we choose a variable for assignment. Similar to the tree search for solving CSPs and the Branch and Bound search for solving COPs, we allow the B&B to invoke consistency enforcing algorithms for WCSPs, hoping to reduce the search space. Line 6 shows the function `LocalConsistency(P, bound)` for calling these enforcing algorithms, and we will introduce them in later sections. After choosing a value v of variable x_i for an assignment, the function `P[xi = v]`(line 7) is invoked to return the subproblem obtained after the assignment.

When a value v is assigned to variable x_i, all constraints $C[S]$ where $x_i \in S$ can be reduced to constraints $C[S'], S' = S - \{x_i\}$. If $C[S]$ returns costs c for a set of assignments $l[S]$, we transform $C[S]$ into $C[S']$ returning the same costs c for $l[S']$. The function `LookAhead(Ps, xi = v)` (line 8) finds all constraints $C[S]$ in the WCSP `Ps` where `xi` $\in S$, and for each of them, the function: 1) Transform $C[S]$ into the $C[S']$ where $S' = S - \{$`xi`$\}$, if $C[S']$ does not exist before the transformation; 2) Transform $C[S]$ and merge the resulting constraint with $C[S']$, if $C[S']$ exists before the transformation. Figure 2.12 shows the function `LookAhead(Ps, xi = v)` for the transformation assuming there are only unary and binary constraints. We can derive functions for higher arity constraints similarly.

The function `cons(Ps,bound)` is used to check whether the costs returned by $C_\varnothing$ of the WCSP `Ps` is greater than or equal to `bound`. If the function returns

```
1  function BranchAndBound(P, bound):
2    if getUnassignedVar(P) == bot: return cost(P)
3    else:
4      xi = getUnassignedVar(P)
5      for v in Di:
6        LocalConsistency(P, bound)
7        Ps = P[xi = v]
8        LookAhead(Ps, xi = v)
9        if cons(Ps,bound):
10         cost = BranchAndBound(Ps, bound)
11         if cost < bound: bound = cost
12       P[xi != v]
13     return bound
```

Figure 2.11: Branch and Bound for WCSPs

```
function LookAhead(Ps, xi = v):
  C_null += Ci(v)
  remove Ci
  for j in [1..n]:
    if Cij in C:
      for u in Dj:
        Cj(u) += Cij(v,u)
      remove Cij
```

Figure 2.12: LookAhead for WCSPs with only unary and binary constraints

true, the cost of the sub-problem `Ps` must be greater than or equal to the bound, and the search will not explore `Ps`; otherwise, the search recursively calls itself to find the cost for the sub-problem obtained. We update the bound if the costs for a sub-problem is lower than the bound (line 11).

Figure 2.13 shows the search tree for solving Example 5 by using the algorithm in Figure 2.11. We denote sub-problems in the search tree by the constraint graph similar to Figure 2.10. We mark sub-problems $\mathcal{P}$ with $C_\varnothing > bound$ by 'Fail' in the figure, as these sub-problems will not be further explored by the search. Once the search finds a feasible complete assignment with a better costs, e.g. $\{x_1 = a, x_2 = b, x_3 = a\}$, the bound is updated. The remaining parts of the search seek better feasible complete assignments. The last found feasible complete assignment is the optimal solutions.

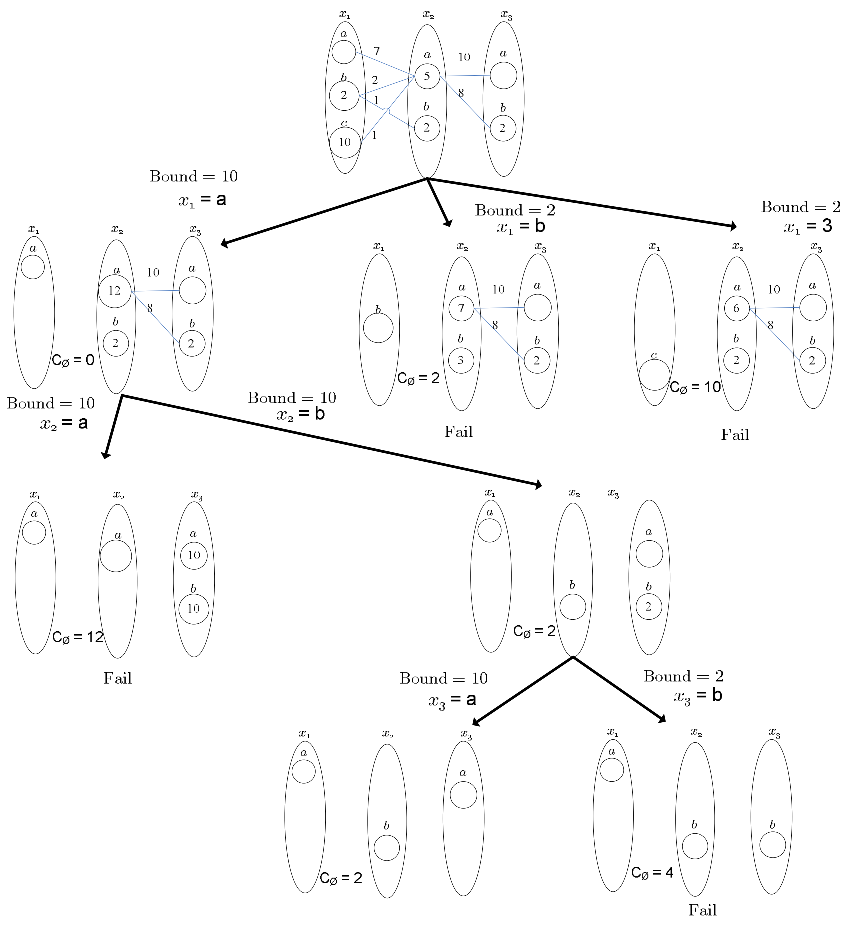

Figure 2.13: Branch and Bound for Example 5

2.2.2 Local Consistencies for WCSPs

Reasons for us to define consistency notions and enforce consistency algorithms in WCSPs are the same as in CSPs. If we perform reasonings for WCSP sub-problems at tree nodes to infer prunings or backtracks, we can reduce the search space. Tree search algorithms traversing smaller search trees will be faster than larger search trees. Consistency enforcing algorithms in WCSPs aim to extract pruning informations by transferring costs between constraints. By transferring costs, we can extract the lower bound of a WCSP as well as inferring values not leading to solutions. In order to define consistency notions, we need to define equivalence of WCSPs.

Definition 2.7. *Given two WCSPs $\mathcal{P}_1(\mathcal{X}, \mathcal{D}_1, \mathcal{C}_1, k)$ and $\mathcal{P}_2(\mathcal{X}, \mathcal{D}_2, \mathcal{C}_2, k)$. The WCSP*

$\mathcal{P}_1$ is equivalent to another WCSP $\mathcal{P}_2$ if: 1) they have the same set of feasible complete assignments, and 2) the cost for each feasible complete assignment is the same on both $\mathcal{P}_1$ and $\mathcal{P}_2$.

Definition 2.8. *Given a local consistency α. An α-enforcing algorithm is to transform a WCSP $\mathcal{P}$ into $\mathcal{P}'$ such that $\mathcal{P}'$ is α and equivalent to P.*

Intuitively, a WCSP is equivalent to another WCSP if they have the same set of feasible complete assignments with the same costs. We restrict consistency enforcing algorithms not to prune values which may lead to feasible solutions.

Node Consistency

Definition 2.9. *[23] For a WCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$, a value u of variable $x_i \in \mathcal{X}$ is node consistent(NC*) iff:*

$$C_\varnothing \oplus C_i(u) < k$$

A variable x_i is node consistent (NC) iff all its values are node consistent, and*

$$\exists v \in D_i \text{ s.t. } C_i(v) = 0$$

WCSP $\mathcal{P}$ is node consistent (NC) iff all its variables are node consistent.*

We want to identify values which are not node consistent, as these values must not participate in any feasible complete assignments. For a variable to be node consistent, in addition to the usual requirement requiring all its values to be node consistent, we further require there exists a value for the variable such that its unary costs is 0. This unusual requirement relates to the fact that we can transfer costs among constraints. For a unary constraint C_i on variable x_i, the minimum costs C_i incur is equal to $\min_{v \in D_i} C_i(v)$, or we alternatively write a shorter notation $\min C_i$. The cost $\min C_i$ is guaranteed to incur, for any value $v \in D_i$ variable x_i takes. If the minimum costs of a unary constraint is larger than zero, we can increase the lower bound $C_\varnothing$, and deduct the minimum costs from the unary constraint. This can further cause values of other variables not node consistent. Suppose we transform a CSP $\mathcal{P}$ into a WCSP $\mathcal{P}'$. NC* on $\mathcal{P}'$ collapses to NC on $\mathcal{P}$. We can easily observe $\mathcal{P}$ is NC iff $\mathcal{P}'$ is NC*.

Example 6. *For the WCSP in Example 5, value c of variable x_1 is not node consistent. Therefore, variable x_1 is not node consistent. Variable x_2 is not node consistent as none of the values v has a unary cost of 0, i.e. $\forall v \in D_2, C_2(v) > 0$. Therefore, the WCSP is not node consistent. Figure 2.14 shows a WCSP which is NC*, and is equivalent to the WCSP in Example 5.*

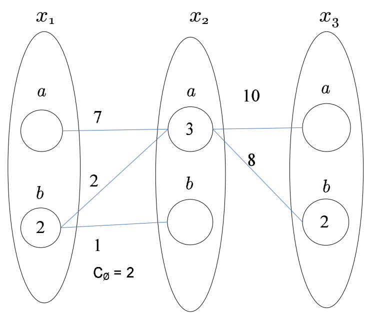

Figure 2.14: WCSP for Example 6

To transform the WCSP in Example 5 which is not NC* to the WCSP in Example 6, we need an operation called projection [23] to preserve the equivalence of WCSPs. We now define projections for unary constraints, called unary projections.

Definition 2.10. *A unary projection of a cost c from C_i to $C_\varnothing$ where $0 \leq c \leq \min_{v \in D_i} C_i(v)$ is defined as an operation transforming $C_\varnothing$ and C_i to $C_\varnothing^\dagger$ and $C_i^\dagger$ respectively, s.t.:*

$$C_\varnothing^\dagger = C_\varnothing \oplus c, \text{, and}$$

$$\forall v \in D_i, C_i^\dagger(v) = C_i(v) \ominus c$$

After defining unary projections, we give the enforcing algorithm `W-NC*` in Figure 2.15 to enforce NC*. The algorithm uses a function `unaryProject(Ci)`

```
1 function W-NC*(P)
2   for i in [1..n]:
3     unaryProject(Ci)
4   for i in [1..n]:
5     for v in Di:
6       if C_null + Ci(v) >= k:
7         prune(xi,v)
8 function unaryProject(Ci)
9   minCost = k
10  for u in Di:
11    if Ci(u) < minCost: minCost = Ci(u)
12  for u in Di:
13    Ci(u) = Ci(u) - minCost
14  C_null = C_null + minCost
```

Figure 2.15: Enforcing algorithm for NC*

to perform unary projection of a cost $\min$ `Ci` from the unary constraint `Ci` to $C_\varnothing$. The function is called n times to perform unary projections on all unary constraints. After calling this function, it is easy to see for all variables x_i, there must exists a value v s.t. $C_i(v) = 0$. At the last stage, we prune all values of all variables which are not node consistent (for loop in line 4 to 7).

Arc Consistency

Definition 2.11. *[23] For a WCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$, a value u of variable $x_i \in \mathcal{X}$ is arc consistent(AC*) with respect to constraint C_{ij} iff:*

$$\exists v \in D_j, s.t. C_{ij}(u, v) = 0.$$

We called the value v a support of value u. Variable x_i is arc consistent(AC) iff it is NC*, and all its values are arc consistent w.r.t. every binary constraint covering on x_i. The WCSP $\mathcal{P}$ is arc consistent(AC*) iff all its variables are arc consistent.*

A value $u \in D_i$ of a variable x_i w.r.t. any binary constraint C_{ij} must incur a cost of $\min_{v \in D_j} C_{ij}(u, v)$, if u is chosen for assignment. We want to extract these minimum costs for all associated binary constraints, and combine with the cost $C_i(u)$. To acheive the goal, we require for each value u of variable x_i w.r.t. a binary constraint C_{ij}, there must be a support. In addition, if unary costs $C_i(u)$ is increased, a NC* WCSP may become not NC*. Therefore, we further require an AC* WCSP to be NC*. Suppose we transform a CSP $\mathcal{P}$ into a WCSP $\mathcal{P}'$. Similar to NC*, AC* on $\mathcal{P}'$ collapses to AC on $\mathcal{P}$. We can easily observe $\mathcal{P}$ is AC iff $\mathcal{P}'$ is AC*.

Example 7. *For the WCSP in Figure 2.14, value b of variable x_1 and value a of variable x_2 is not arc consistent (AC*). Variable x_1 and x_2 is not arc consistent (AC*), and therefore, the WCSP is not arc consistent (AC*). Figure 2.16 shows a WCSP which is AC* (therefore NC*), and is equivalent to the two WCSPs in Figure 2.10 and Figure 2.14.*

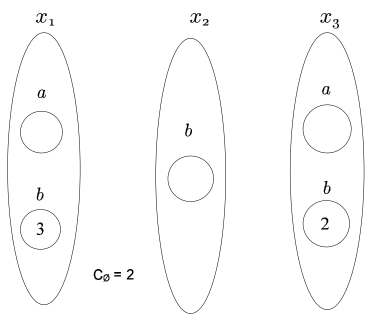

Figure 2.16: WCSP for Example 7

In order to transform the WCSP in Example 6 which is not AC* to the AC* WCSP in Figure 2.16, we need to use projections [23]. We now define binary projections for binary constraints.

Definition 2.12. *A binary projection of a cost c from C_{ij} to C_i for value u of x_i where $0 \leq c \leq \min_{v \in D_j} C_{ij}(u,v)$ is defined as an operation transforming C_i and C_{ij} to $C_i^\dagger$ and $C_{ij}^\dagger$ respectively, s.t.:*

$$\begin{aligned}
C_i^\dagger(u) &= C_i(u) \oplus c \\
\forall a \in D_i - \{u\}, C_i^\dagger(a) &= C_i(a) \\
\forall v \in D_j, C_{ij}^\dagger(u,v) &= C_{ij}(u,v) \ominus c \\
\forall a \in D_i - \{u\}, \forall v \in D_j, C_{ij}^\dagger(a,v) &= C_{ij}(a,v)
\end{aligned}$$

We can find supports for values of x_i from values of x_j w.r.t. C_{ij} by using binary projections. Figure 2.17 shows the resulting WCSP in Figure 2.14 after finding supports for all values of all variables using binary projections. We can achieve the NC* and AC* WCSP in Figure 2.16 after applying the NC* enforcing algorithm. We are now ready to give the enforcing algorithm `W-AC*3` [23] for AC*.

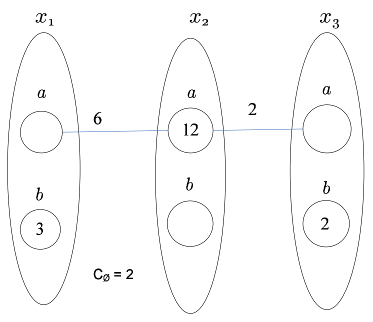

Figure 2.17: WCSP after binary projections

The enforcing algorithm is based on the AC enforcing algorithm `AC3` for CSPs. Similar to the AC enforcing algorithm for CSPs, we maintain a propagation queue `Q`. Initially, all variables are placed into the queue. Variables with values being pruned will also be placed into the queue. If the queue is not empty, we retrieve a variable x_j by the function `pop(Q)` (line 4). All binary constraints C_{ij} associated with x_j will be checked for supports by calling the function `FindSupportAC3(Cij)` (line 7).

The loop from line 18 to line 24 in the function is essentially equivalent to performing binary projections for all values u of x_i with a cost $\min_{v \in D_j} C_{ij}(u, v)$ from C_{ij} to C_i. After finding supports for each value u of variable x_i, the unary costs $C_i(u)$ may be increased. This may cause variable x_i not NC*. Therefore, we need to call the function `unaryProject(Ci)` (line 26) introduced in the enforcing algorithm of NC*. Function `FindSupportAC3(Cij)` returns true if $C_\varnothing$ is increased after calling the function.

In the final step, we need to enfore NC*. Function `FindSupportsAC3(Cij)` ensures one of the condition $\exists v \in D_i, s.t. C_i(v) = 0$ in NC*. We are left to prune values v which cannot satisfy the other condition: $C_\varnothing \oplus C_i(v) < k$. This is done by

```
1 function W-AC*3(P)
2   Q = {1,2,...,n}
3   while Q != null:
4     j = pop(Q)
5     flag = false
6     for Cij in C
7       flag = flag || FindSupportsAC3(Cij)
8       PruneVar(xi,Q)
9     if flag == true:
10      for i in [1..n]:
11        PruneVar(xi,Q)
12 function PruneVar(xi,Q)
13  for u in Di:
14    if C_null + Ci(u) >= k:
15      prune(xi,u)
16      addQueue(Q,i)
17 function FindSupportAC3(Cij)
18  for u in Di:
19    minCost = k
20    for v in Dj:
21      if Cij(u,v) < minCost: minCost = Cij(u,v)
22    Ci(u) = Ci(u) + minCost
23    for v in Dj:
24      Cij(u,v) = Cij(u,v) - minCost
25  original = C_null
26  unaryProject(Ci)
27  return original != C_null
```

Figure 2.18: Enforcing algorithm for AC*

calling the function `PruneVar(xi,Q)`. There are two cases for calling this function. After finding supports for a value v of variable x_i, costs $C_i(v)$ may increases. If value v does not satisfy the condition $C_\varnothing \oplus C_i(v) < k$, we can prune this value by calling `PruneVar(xi,Q)` in line 8. In the other case, if $C_\varnothing$ is increased, values u of variables $x_l, l \neq i$ may not satisfy the condition $C_\varnothing \oplus C_l(u) < k$. We need to prune these values by calling `PruneVar(xi,Q)` in line 11.

We need to re-check supports for all binary constraints on x_i if values of x_i is pruned, and therefore, we place x_i into the queue `Q` again (line 16). This is similar to AC in CSPs.

The time complexity of `W-AC*3` is $O(n^2d^2 + ed^3)$, where n is the number of variables, d is the maximum domain size, and e is the number of constraints.

2.3 Quantified Constraint Satisfaction Problems

In recent years, we are interested in exploring quantified constraint satisfaction problems (QCSPs). QCSPs can solve model checking problems, adversary games and planning problems under uncertainty conditions. QCSPs generalize CSPs by allowing variables to take either existential $\exists$ or universal $\forall$ quantifiers. This is similar to how Quantified Boolean Formula (QBF) [19] generalizing SAT by allowing variables to take either existential or universal quantifiers. For a QCSP, we are interested to find whether there exists assignments for existential variables, for all values in universal variables, such that the constraints are satisfied. This generalization raises the hardness of the problem from NP-complete to PSPACE-complete [15].

A *quantified constraint satisfaction problem* [31] (QCSP) $\mathcal{P}$ is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q})$, where $\mathcal{X} = (x_1, x_2, \ldots, x_n)$ is an ordered sequence of variables, $\mathcal{D} = (D_1, D_2, \ldots, D_n)$ is an ordered sequence of finite domains, $\mathcal{C} = \{C_1, C_2, \ldots, C_e\}$ is a set of constraints, and $\mathcal{Q} = (Q_1, Q_2, \ldots, Q_n)$ is a quantifier sequence in which each Q_i is either an existential quantifier $\exists$ or a universal quantifier $\forall$. A *constraint* $C[S] \in \mathcal{C}$ consists of a sequence S of variables s.t. S is a subsequence of $\mathcal{X}$. $C[S]$ has an associated set $L(C[S])$ of *tuples* which specify the allowed combinations of values for variables in S. The *arity* of the constraint $C[S]$ is defined as the number of variables in the sequence S, i.e. $|S|$. We reuse the notion of unary constraints, binary constraints, and n-ary constraints defined in Section 2.1. Similar to previous sections, we denote a unary constraint $C[(x_i)]$ by C_i, and a binary constraint $C[(x_i, x_j)]$ by C_{ij}. We write $x_i = v_i$ as an *assignment* assigning value $v_i \in D_i$ to variable $x_i \in \mathcal{X}$, i.e. variable x_i taking value v_i. Let $l = (x_{i_1} = v_{i_1}, x_{i_2} = v_{i_2}, \ldots, x_{i_m} = v_{i_m})$ be a sequence of assignments, where the sequence is ordered according to the variable ordering defined in $\mathcal{X}$. If l contains the assignment $x_i = v_i$, we abuse set notations by saying $x_i = v_i \in l$. If l contains a subsequence S, we abuse set notations by saying $S \subseteq l$. We abuse terminology by saying the *scope* of l ($scope(l)$) to be the subsequence of variables involved in l, i.e. $scope(l) = (x_{i_1}, x_{i_2}, \dots, x_{i_m})$. We sometimes denote l by a tuple $(v_{i_1}, v_{i_2}, \dots, v_{i_m})$ if the associated variable for each coordinate in the tuple is clear. A *complete assignment* is an l where $scope(l) = \mathcal{X}$. A *partial assignment* $l[S]$ is a subsequence of l, such that if l contains an assignment $x_i = v_i$ and $x_i \in S$, $l[S]$ contains $x_i = v_i$. A sequence of assignments l *satisfy* a constraint $C[S]$, where S is a subsequence of $scope(l)$, if $l[S]$ specifies an allowed combinations of values the sequence of variables S can take, i.e. $l[S] \in L[C[[S]]$. Let firstx($\mathcal{P}$) returns the first unassigned variable in the variable sequence. If there are no such variables, it returns $\perp$. The *semantics* of a QCSP $\mathcal{P}$ is defined recursively as follows:

- In case $\text{firstx}(\mathcal{P}) = \perp$, if all constraints $C[S] \in \mathcal{C}$ are satisfiable, $\mathcal{P}$ is *satisfiable*; and if any constraint fails, $\mathcal{P}$ is *unsatisfiable*.

- Otherwise, let $\text{firstx}(\mathcal{P}) = x_i$. If $Q_i = \exists$ then $\mathcal{P}$ is *satisfiable* iff there exists a value $a \in D_i$ such that the simplified problem $\mathcal{P}$ with a assigned to x_i is satisfiable. If $Q_i = \forall$ then $\mathcal{P}$ is *satisfiable* iff for all values $a \in D_i$ the simplified problem $\mathcal{P}$ with a assigned to x_i is satisfiable.

Example 8. *Given a QCSP $\mathcal{P}$ with the ordered sequence of variables $\mathcal{X} = (x_1, x_2, x_3)$, domains $D_1 = \{2, 3, 5\}$, and $D_2 = D_3 = \{4, 5\}$, the set of constraints $\mathcal{C} = \{x_1 > 2, x_1 \neq x_2, x_2 \neq x_3\}$, and the sequence of quantifiers $(\exists, \forall, \exists)$. We usually write the QCSP $\mathcal{P}$ as follows:*

$$\exists x_1 \forall x_2 \exists x_3 \; s.t. \; x_1 > 2, x_1 \neq x_2, x_2 \neq x_3$$

We are asking does there exists a value for x_1, for all values of x_2, there exists a value of x_3 such that the three constraints, $x_1 > 2, x_1 \neq x_2, x_2 \neq x_3$, are satisfied.

In Example 8, we can see that there exists value 3 of x_1, for value 4 of x_2, there exists value 5 of x_3, such that all the three constraints are satisfied. If the value of x_2 is now changed to 5, there exists also a value 4 of x_3 such that all the constraints are satisfied. We can conclude the above QCSP is satisfiable. In order to perform the above reasonings easier, we need to formalize solutions for QCSPs.

In the QCSP literature, there are different notations for solutions [11, 31, 5]. The notation we use mainly follows from Bordeaux et.al., with slight modifications. A complete assignment t is a *feasible assignment* to a QCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q})$ if t satisfies all constraints in $\mathcal{C}$. A *strategy* is a family $\{s_i | Q_i = \exists, 1 \leq i \leq n\}$ of functions of the following type: for each existential variable $x_i \in \mathcal{X}$, the function s_i associates to each tuple $t \in \times_{j=1,Q_j=\forall}^{i-1} \mathcal{D}_j$ a value $v_i \in \mathcal{D}_i$. Intuitively, we can view each function s_i specifies which value $v_i \in D_i$ should be assigned to x_i for a combination of values taken by the preceding universal variables. The function s_i is a constant function if there are no universal variables preceding variable x_i. A *scenario* t of a strategy s is a complete assignment such that for each value $t[(x_i)]$ assigning to an existential variable x_i, the value follows according to the strategy s_i. We say $t[(x_i)]$ follows according to the strategy s_i if $t[(x_i)]$ is the value specifies by s_i for the tuple $\times_{j=1,Q_j=\forall}^{i-1} \{t[(x_j)]\}$. We are usually interested in knowing the set of possible scenarios of a strategy s. If all possible scenerios of a strategy s are feasible assignments, we called the strategy a *winning strategy/solution*. A QCSP is satisfiable iff there exists a *winning strategy/solution*.

Bordeaux et al. [11] use adversarial viewpoint to understand quantifier alternations of QCSPs. We can view a QCSP as a game, and there are two players interacting in the game. One of the players chooses values for existential variables aiming to satisfy all constraints, while the other player chooses values for universal variables aiming to falsify at least one constraint. The game is played in a turn-based manner, and values for variable x_i will be chosen in each turn i. Therefore, the game is played according to the variable sequence. Treating a QCSP as a game leads to game-theoretic terminology. In particular, the notion of strategies and winning strategies in games fits well to the semantic of QCSP.

Example 9. *We give a strategy $s_A = \{s_1, s_3\}$ for the QCSP in Example 8 as an example. The function s_1 is a constant function giving value $3 \in D_1$. The function s_3 maps to value $5 \in D_3$ ($4 \in D_3$ resp.) if variable x_2 takes $4 \in D_2$ ($5 \in D_2$ resp.). The two outcomes of strategy s_A are $(x_1 = 3, x_2 = 4, x_3 = 5)$ and $(x_1 =$*

$3, x_2 = 5, x_3 = 4)$. *As all the outcomes of strategy s_A are feasible assignments, s_A is a winning strategy. We can construct another strategy $s_B = \{s_1, s_3\}$ similar to s_A. The function s_1 is a constant function giving value $5 \in D_1$, and the function s_3 maps to value $5 \in D_3$ ($4 \in D_3$ resp.) if variable x_2 takes $4 \in D_2$ ($5 \in D_2$ resp.). As one outcome $\{x_1 = 5, x_2 = 5, x_3 = 4\}$ of strategy s_B is not a feasible assignment, s_B is not a winning strategy.*

We can use And-Or trees based on labeling trees for CSPs to explain the semantics and solution space of QCSPs. The root node of an And-Or tree is at level 1, and the level of a node is equal to the level of its parents plus one. Nodes at level i are labelled as 'Or' nodes ('And' nodes resp.) if $Q_i = \exists$ ($Q_i = \forall$ resp.).

Let $\mathcal{P}[x_i = v_i]$ denotes the sub-problem obtained from the QCSP $\mathcal{P}$ by assigning v_i to x_i. Suppose $Q_1 = \forall$. If $\mathcal{P}$ is satisfiable, then for all $v_1 \in D_1$, $\mathcal{P}[x_1 = v_1]$ must be satisfiable. We label the root node as an 'And' node, indicating all the sub-problems $\mathcal{P}[x_1 = v_1]$ must be satisfiable for $\mathcal{P}$ to be satisfiable. On the other hand if $Q_1 = \exists$. If $\mathcal{P}$ is satisfiable, then there exists a sub-problem $\mathcal{P}[x_1 = v_1]$ where $v_1 \in D_1$ which is satisfiable. We label the root node as an 'Or' node, indicating if a sub-problem $\mathcal{P}[x_1 = v_1]$ is satisfiable, then $\mathcal{P}$ is satisfiable. Followed by the same reasonings, we can infer labelings for all the remaining nodes inductively. Leaf nodes in the tree represent complete assigned sub-problems. If these problems are satisfiable, then the associated complete assignments are feasible assignments. We can easily infer satisfiability of a QCSP by following its And-Or tree, viewing from bottom to top.

Figure 2.19 shows the And-Or tree for the QCSP in Example 8. We label all feasible assignments. We also show the outcomes for the two strategies s_A and s_B in Example 9. With the help of the And-Or tree, we can see a strategy of a QCSP specifies a sub-graph G of the associated And-Or Tree. The sub-graph G contains the root node of the And-Or Tree. If G contains an 'And' node, all its children are contained in the graph. Suppose G contains an 'Or' node at level i, representing a partial assignment $p = (x_1 = v_1, \ldots, x_{i-1} = v_{i-i})$. Let the

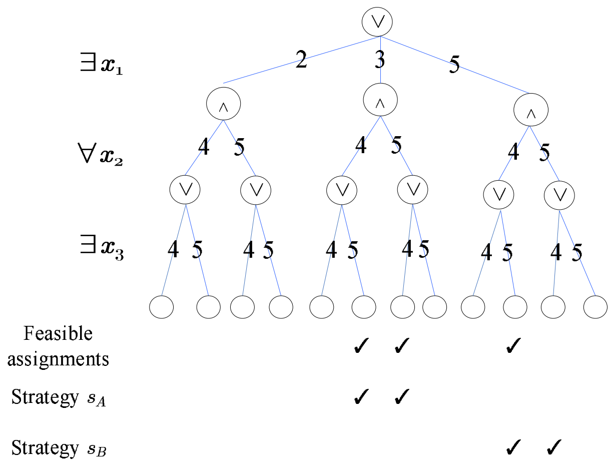

Figure 2.19: And-Or Tree for Example 8

sequence S be a maximum subsequence of $(x_1, \ldots, x_{i-1})$, such that each variable in the subsequence must be a universal variable. The function s_i in the strategy must specify which value $v_i \in D_i$ for the tuple of values $p[S]$, if $i > 1$ and S is not an empty sequence. If $i = 1$ or S is an empty sequence, the function s_i must be a constant function specifying which value $v_i \in D_i$. The child node representing the assignment $x_i = v_i$ must contain in G. As the sub-graph G is unique for each strategy, we can represent strategies using sub-graphs.

Theorem 2.13. *A CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ can be transformed by Karp reduction [2] to an equivalent QCSP $\mathcal{P}' = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}')$ where all quantifiers in $\mathcal{Q}'$ are $\exists$.*

Proof. Suppose $\mathcal{P}'$ is satisfiable. The winning strategy $s = \{s_1, \ldots, s_n\}$ of $\mathcal{P}'$ is a set of constant functions, where each constant function s_i indicates which value v_i the variable x_i should take. By definition of winning strategy, the only outcome of strategy s is a complete assignment satisfying all the constraints in $\mathcal{C}$. Therefore, $\mathcal{P}$ must be satisfiable.

Suppose $\mathcal{P}'$ is unsatisfiable. This implies there does not exist a strategy such that all its outcomes are feasible assignments. As all quantifiers of $\mathcal{P}'$ are existential quantifiers, all possible strategies must be a set of constant functions s_i indicating which value v_i should x_i takes. The outcome of a strategy must be a complete assignment. Finally, we can conclude there does not exist any complete assignment satisfying all the constraints $\mathcal{C}$. Otherwise, we can construct a strategy which is winning. $\mathcal{P}$ must be unsatisfiable. □

2.3.1 Backtracking Tree search

To solve a QCSP, we can modify the backtracking tree search for CSPs to traverse the associated And-Or Tree. Figure 2.20 shows a high level abstraction of the tree search for QCSPs. The tree search algorithm mainly follows the tree search used by QCSP-Solve [21]. If the node is an 'Or' node, the search is required to traverse all its children in order to prove the associated sub-problem is not satisfiable. If any child returns satisfiable, the search can infer the associated sub-problem is satisfiable. The for loop in Lines 7 to 14 shows how the search traverses 'Or' nodes. On the other hand if the node is an 'And' node, the search is required to traverse all its children in order to prove the associated sub-problem is satisfiable. If any child returns unsatisfiable, the search can infer the associated sub-problem is not satisfiable. The for loop in Lines 16 to 23 shows how the search traverses 'And' nodes. We can easily see the two loops are symmetric to each other.

The search uses a function `firstx(P)` to return the first unassigned variables for the QCSP `P`. If there are no variables unassigned, the function returns `bot`. The function `cons(P)` is used to check whether the current complete assignment for `P` satisfies all constraints. It returns `true` if all constraints are satisfied, otherwise it returns `false`. We reuse the two functions `P[xi = v]` and `P[xi != v]` defined in Section 2.1.

The function `localConsistency(P)` (line 8 and 17) is used to maintain consistencies during search. We will introduce consistency notions in subsequent sections. To solve QCSPs, we can use different kinds of forward checking algorithms. In particular, QCSP-Solve implements two kinds of forward checking algorithms FC0 [27] and FC1 [27] for QCSPs with binary constraints only. Function `FC(P)` in line 10 and 19 is used to invoke these forward checking algorithms.

```
function backtrackingSearch(P):
  if firstx(P) == bot:
    return (cons(P))? true: false
  if firstx(P) != bot:
    xi = firstx(P)
    if Qi == exists:
      for v in Di:
        localConsistency(P)
        Ps = P[xi = v]
        if FC(Ps):
          if backtrackingSearch(Ps) == true:
            return true
        P[xi != v]
      return false
    if Qi == forall:
      for v in Di:
        localConsistency(P)
        Ps = P[xi = v]
        if FC(Ps):
          if backtrackingSearch(Ps) == false:
            return false
        P[xi != v]
      return true
```

Figure 2.20: Backtracking Tree Search for QCSP

Figure 2.21 shows the search tree for Example 8 without applying preprocessing, enforcing local consistencies, and using forward checking algorithms. We assume the tree search checks the satisfiability of a constraint only when a constraint is fully assigned. We represent sub-problems for tree nodes by its constraints. We mark tree nodes by ‘Fail’ if the corresponding partial assignments violate at least one constraint. Feasible assignments are marked by ‘Feasible’.

2.3.2 Consistencies for QCSPs

Similar to CSPs and WCSPs, defining consistency notions help to extract pruning informations. We redefine the notions for equivalence and enforcing algorithms for QCSPs.

Definition 2.14. *Given two QCSPs $\mathcal{P}_1(\mathcal{X}, \mathcal{D}_1, \mathcal{C}_1, Q)$ and $\mathcal{P}_2(\mathcal{X}, \mathcal{D}_2, \mathcal{C}_2, Q)$. The QCSP $\mathcal{P}_1$ is equivalent to another QCSP $\mathcal{P}_2$ if they have the same set of winning*

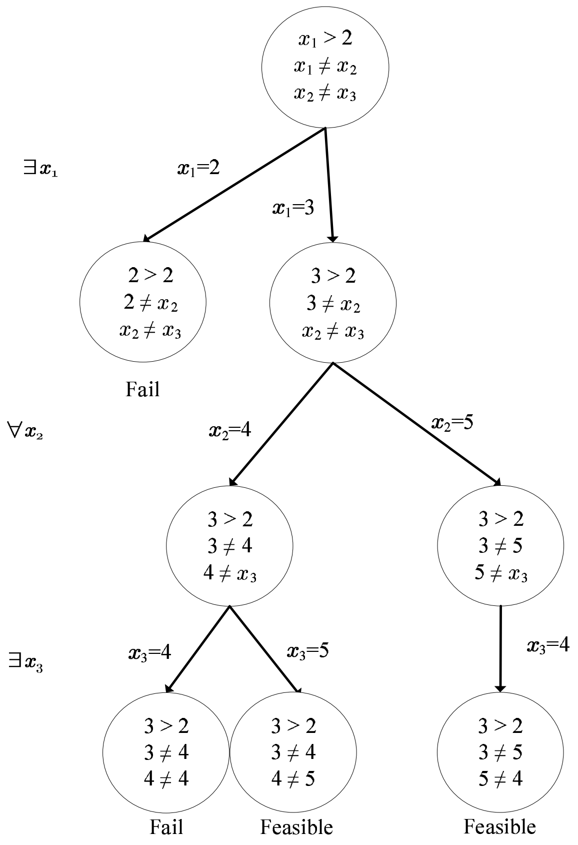

Figure 2.21: Search Tree for Example 8

strategies.

Definition 2.15. *Given a local consistency α. An α-enforcing algorithm is to transform a QCSP $\mathcal{P}$ into $\mathcal{P}'$ such that $\mathcal{P}'$ is α and equivalent to P.*

For a value $v_i \in D_i$ of an existential variable $x_i \in \mathcal{X}$, if the assignment $x_i = v_i$ does not satisfy the unary constraint C_i, we can prune v_i. The reason is similar to node consistency (NC) in CSPs. If x_i is a universal variable, and the assignment $x_i = v_i$ does not satisfy C_i, the QCSP is trivially unsatisfiable. We can prune all values $v_i \in D_i$ to indicate the QCSP is unsatisfiable. Similar to CSPs, all unary constraints in the QCSP can be trivially removed after preprocessing. Based on the above intuition, we can define node consistency (NC) and its enforcing algorithm as follows.

Definition 2.16. *A value $v_i \in D_i$ of a variable x_i is node consistent (NC) if $x_i = v_i$ satisfies unary constraint C_i. A variable x_i is node consistent (NC) if all its values are NC. A QCSP is node consistent (NC) if all its variables are NC. If a value $v_i \in D_i$ of an existential variable x_i is not NC, we prune v_i to enforce NC. If a value of a universal variable x_i is not NC, the QCSP must be unsatisfiable.*

NC conditions for QCSPs are the same as CSPs. The enforcing algorithm for NC in CSPs cannot be reused as it is unsound for QCSPs. We have to follow the pruning conditions/filtering conditions. In particular, we cannot mix up the two pruning conditions for enforcement algorithms. If a value v_i of a universal variable is not NC, and we prune v_i. We may transform an unsatisfiable QCSP into a satisfiable QCSP. Figure 2.22 shows the NC enforcing algorithm for QCSPs, by following the pruning conditions. Similar to the NC enforcing algorithm for CSPs, we use a function `cons(Ci,vi)` to check whether value $v_i \in D_i$ satisfies the unary constraint C_i, and a function `prune(xi,vi)` to prune value v_i of variable x_i. We also introduce a function `backtrack()` to perform backtrack.

Similar to CSPs, it is safe to remove all unary constraints after enforcing NC. The preprocessing algorithm in the tree search always enforce NC before search.

```
function QNC(P):
  for xi in [1..n]:
    if Ci exists:
      for vi in Di:
        if cons(Ci,vi) == false:
          if Qi == exists:
            prune(xi,vi)
          if Qi == forall:
            backtrack()
            return
```

Figure 2.22: Node Consistency Enforcing Algorithm for QCSP

For a binary constraint $C_{ij}, i < j$ in QCSPs, they can be splitted into four types depending on the quantifiers of variable x_i and x_j. Type $C_{\exists x_i \exists x_j}$ represents binary constraints where $Q_i = \exists$ and $Q_j = \exists$, type $C_{\exists x_i \forall x_j}$ represents binary constraints where $Q_i = \exists$ and $Q_j = \forall$, type $C_{\forall x_i \exists x_j}$ represents binary constraints where $Q_i = \forall$ and $Q_j = \exists$, and type $C_{\forall x_i \forall x_j}$ represents binary constraints where $Q_i = \forall$ and $Q_j = \forall$.

Definition 2.17. *We define arc consistencies (AC) [27] for each type of constraints:*

- *For $C_{\exists x_i \exists x_j}$*

  This is the case for classical CSPs. Binary constraint C_{ij} is AC iff all values in D_i are supported by at least one value $v_j \in D_j$. If C_{ij} is not AC, there must exists values $v_i \in D_i$ having no supports from all values in D_j. To enforce AC, we prune these values v_i. The QCSP is unsatisfiable if D_i is empty.

- *For $C_{\exists x_i \forall x_j}$*

  Binary constraint C_{ij} is AC iff all values in D_i are supported by all values $v_j \in D_j$. If C_{ij} is not AC, there must exists values $v_i \in D_i$ not having a support from at least one of the values in D_j. To enforce AC, we prune these values v_i. The QCSP is unsatisfiable if D_i is empty.

- *For $C_{\forall x_i \exists x_j}$*

  Binary constraint C_{ij} is AC iff all values in D_i are supported by at least one *value $v_j \in D_j$. If C_{ij} is not AC, there must exists values $v_i \in D_i$ having no supports from all values in D_j. If such value $v_i \in D_i$ is found, the problem must be unsatisfiable.*

- *For $C_{\forall x_i \forall x_j}$*

  Binary constraint C_{ij} is AC iff all values in D_i are supported by all values $v_j \in D_j$. If C_{ij} is not AC, there must exists values $v_i \in D_i$ not having a support from at least one of the values in D_j. If such value $v_i \in D_i$ is found, the problem must be unsatisfiable.

We can see the AC conditions for $C_{\exists x_i \exists x_j}$ and $C_{\forall x_i \exists x_j}$ (, and also $C_{\exists x_i \forall x_j}$ and $C_{\forall x_i \forall x_j}$) are the same, depending on quantifier of variable x_j (Q_j). However, the pruning conditions for enforcement algorithms depend on the quantifier Q_i. If $Q_i = \exists$, we will prune values of x_i. Otherwise, we prune all values of x_i or perform backtrack. There is one interesting observation for the two types of binary constraints $C_{\exists x_i \forall x_j}$ and $C_{\forall x_i \forall x_j}$. If all binary constraints of these two types are AC, we can safely remove these constraints. We always enforce AC for these two types of constraints during preprocessing.

Example 10. *Given a QCSP $\mathcal{P}$ with the ordered sequence of variables $\mathcal{X} = (x_1, x_2, x_3)$, domains $D_1 = \{1, 2\}$, $D_2 = \{2, 3, 5\}$, $D_3 = \{4, 5\}$, the set of constraints $\mathcal{C} = \{x_1 \neq x_2, x_2 < x_3\}$, and the sequence of quantifiers $(\exists, \forall, \exists)$. Both constraints are not AC. For constraint $x_1 \neq x_2$, value 2 of x_1 does not have supports from all values of x_2. We can prune 2 of x_1 to make the constraint AC. After pruning, we can safely remove the constraint. For constraint $x_2 \leq x_3$, value 5 of x_2 does not have any support from values of x_3. We can conclude $\mathcal{P}$ is unsatisfiable. If the constraint $x_2 < x_3$ changes to $x_2 \leq x_3$, the consraint is AC. All values of x_2 now have at least one support from values of x_3.*

Example 11. *The QCSP in Example 8 is not NC and AC. If we apply NC and AC in Example 8, we can achieve an equivalent QCSP $\mathcal{P}$ with the ordered sequence of* *variables $\mathcal{X} = (x_1, x_2, x_3)$, domains $D_1 = \{3\}$, and $D_2 = D_3 = \{4, 5\}$, the set of constraints $\mathcal{C} = \{x_2 \neq x_3\}$, and the sequence of quantifiers $(\exists, \forall, \exists)$. Note that we can safely remove constriant $x_1 > 2$ and $x_1 \neq x_2$.*

Figure 2.23 shows a high level abstraction of the AC enforcing algorithm for QCSPs [20], which is based on AC-2001$\backslash$3.1. The algorithm assumes the input QWCSP has no unary constraints, as these constraints can be removed during preprocessing (by enforcing NC once).

The algorithm starts by preprocessing binary constraints of type $C_{\exists x_i \forall x_j}$ and $C_{\forall x_i \forall x_j}$ (for loop in line 3 to 19). These two types of binary constraints will be removed after preprocessing. We store all these two types of constraints in the set S', and use the function `addS(S',Cij)` to add them in S' one by one.

Line 21 shows a function `initializeSupport()` used to initialize the data structure for storing supports. Initially, all values do not have any supports. The while loop in line 22 to 28 resembles the propagation loop in the AC enforcing algorithm for CSPs. Each constraint C_{ij} in the propagation queue `S` will be processed by finding supports for values $v_i \in D_i$ of variable x_i.

The function `Revise(P,Cij)` is the core function used to find supports for values of variable x_i w.r.t. constraint `Cij`, and it returns true if a value $v_i \in D_i$ is being pruned. If any value v_i is pruned and the type of constraint currently processing is $C_{\forall x_i \exists x_j}$, the solver can perform backtracking. The solver can also backtrack if all values in D_i are pruned. Both cases follow according to the pruning conditions in Definition 2.17.

If any value v_i of x_i is pruned, other constraints covering on variable x_i may not be AC. We need to enfore AC for these constraints. The function `add(S,i,j)` (line 28) is then used to add constraints C_{ki} where $k \neq i \wedge k \neq j \wedge C_{ki} \in \mathcal{C} - S'$ to the propagation queue S.

The function `noSupport(vi,Cij)` is used to check whether value `vi` $\in D_i$ of variable x_i has a support from values in D_j w.r.t. the constraint `Cij`. If there are no supports, the function returns true; otherwise, it returns false. In AC-2001$\backslash$3.1, the function first checks whether a support for value `vi` w.r.t the constraint `Cij` was found during past executions of this function. If true, the function immediately returns false. Otherwise, the function searchs supports from D_j, starting from the last found supports. By storing previous found supports, `noSupport(vi,Cij)` reduces time to find supports which have not been pruned.

```
function QAC-2001(P):
  S’ = {}
  for Cij in C:
    if Qi = forall && Qj = forall:
      for vi in Di:
        for vj in Dj:
          if cons(Cij,vi,vj) == false:
            backtrack()
            return
      addS(S’,Cij)
    if Qi = exists && Qj = forall:
      for vi in Di:
        for vj in Dj:
          if cons(Cij,vi,vj) == false:
            prune(xi,vi)
      if emptyDomain(Di):
        backtrack()
        return
      addS(S’,Cij)
  S = {Cij in C} - S’
  initializeSupport()
  while S != null:
    Cij = pop(S)
    if Revise(P,Cij):
      if Qi = forall || emptyDomain(Di):
        backtrack()
        return
      add(S,i,j)
function Revise(P,Cij):
  delete = false
  for vi in Di:
    if noSupport(vi,Cij):
      prune(xi,vi)
      if Qi = forall:
        return true
      delete = true
  return delete
```

Figure 2.23: Arc Consistency Enforcing Algorithm for QCSP

The worst case time complexity for `QAC-2001` is $O(ed^2)$ [20], where e is the number of binary constraints and d is the maximum domain size. It is the same as AC-2001\3.1.

A QCSP $\mathcal{P}$ is called a *binary* QCSP if $\mathcal{P}$ has only unary and binary constraints. If we enforce NC and AC on $\mathcal{P}$, we can safely remove all unary constraints, and all binary constraints of type $C_{\exists x_i \forall x_j}$ and $C_{\forall x_i \forall x_j}$. Therefore, a constraint of a preprocessed binary QCSP is either a binary constraint of type $C_{\exists x_i \exists x_j}$ or a binary constraint of type $C_{\forall x_i \exists x_j}$.

2.3.3 Look Ahead for QCSPs

We now introduce two look ahead algorithms, FC0 and FC1, for QCSPs [27, 21]. FC0 is an algorithm that can discover dead-ends early by forward checking the current variable assignment $x_i = v_i$ against values $v_j \in D_j$ of future existential variables $x_j : j > i \wedge Q_j = \exists$ constrained with current variable. The reason behind is that after preprocessing, we have only binary constraints of type $C_{\exists x_i \exists x_j}$ and type $C_{\forall x_i \exists x_j}$. Variables x_j for these types of constraints are existential variables. Given a preprocessed binary QCSP $\mathcal{P}$ with the current variable assignment $x_i = v_i$. To apply FC0, we check whether all sequences of variable assignments $(x_i = v_i, x_j = v_j)$ where $x_j : j > i \wedge Q_j = \exists, v_j \in D_j$ satisfy the binary constraint C_{ij}. If a sequence of variable assignments violates the respective binary constraint, we prune value v_j of x_j.

FC1 has exactly the same behavior as FC0 if the current variable being assigned is an existential variable. If the current variable x_i is a universal variable and v_i is assigned, FC1 will check not only v_i, but all values in D_i against all future variables $x_j : j > i$ before assigning a specific value to x_i. If a value in D_i causes backtrack, then FC1 performs backtracks. Otherwise, it proceeds by assigning v_i to x_i, and checks whether all sequences of variable assignments $(x_i = v_i, x_j = v_j)$ with $x_j : j > i \wedge Q_j = \exists$ and $v_j \in D_j$ satisfy the binary constraint C_{ij}. If a sequence of variable assignments violates the respective binary constraint, we prune value v_j of

x_j.

Suppose the search is allowed to preprocess the QCSP in Example 8, and to apply the FC1 forward checking algorithm. After preprocessing, we are solving the smaller QCSP in Example 11 which is equivalent to the QCSP in Example 8. Figure 2.24 shows the corresponding search tree.

Apart from apply preprocessing and search with forward checking, we can replace forward checking by algorithms maintaining AC (MAC). In particular, we can devise MAC algorithms based on FC1 called MAC1. If the current variable x_i is an existential variabe and v_i is assigned, MAC1 enforces AC (by running `QAC-2001`) after the assignment is made. When the current variable x_i of a QCSP $\mathcal{P}$ is a universal variable and v_i is assigned, MAC1 not only applies AC for the sub-problem $\mathcal{P}[x_i = v_i]$, but on all sub-problems $\mathcal{P}[x_i = v_j]$ where $v_j \in D_i$. If any of these sub-problems is unsatisfiable, we can backtrack.

There are also other techniques which have been investigated for solving QCSPs. These techniques include: 1) pure value rule [21], 2) solution directed pruning [21]/conflict-based backjumping [3], and 3) symmetry breaking [21]. These techniques can be found in QCSP-Solve.

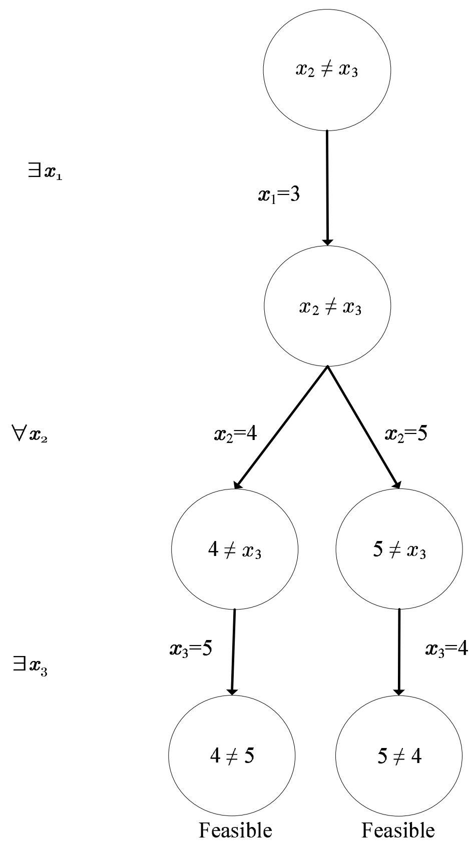

Figure 2.24: Search Tree for Example 11

Chapter 3

Quantified Weighted CSPs

Standard WCSPs are minimization in nature, we aim at also optimizing problems with adversarial conditions, by modeling adversaries using $\max$ quantifiers. A *Quantified Weighted Constraint Satisfaction Problem* (QWCSP) $\mathcal{P}$ is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}, k)$, where $\mathcal{X} = (x_1, \ldots, x_n)$ is defined as an ordered sequence of *variables*, $\mathcal{D} = (D_1, \ldots, D_n)$ is an ordered sequence of finite *domains*, $\mathcal{C}$ is a set of *soft constraints* as in WCSPs, $\mathcal{Q} = (Q_1, \ldots, Q_n)$ is a *quantifiers sequence* where Q_i is either $\max$ or $\min$ associated with x_i, and k is the global upper bound.

We reuse the notions of assignments, scopes of a set of assignments, complete assignments, partial assignments, constraint arities, unary constraints, and binary constraints for WCSPs. Similar to WCSPs, we name constraints in CSPs as hard constraints and constraints in WCSPs as soft constraints to distinguish constraints between CSPs/QCSPs and QWCSPs. We write C_i for the unary constraint $C[\{x_i\}]$ on variable x_i, C_{ij} for the binary constraint $C[\{x_i, x_j\}]$ on variables x_i and x_j, $C_i(d)$ for the cost returned by the unary constraint C_i when d is assigned to x_i, and $C_{ij}(u, v)$ for the cost returned by the binary constraint C_{ij} when u and v is assigned to x_i and x_j respectively.

In a QWCSP, ordering of variables is important. Without loss of generality, we assume variables are ordered by their indices. We define a variable with $\min$ ($\max$ resp.) quantifier to be a minimization variable (maximization variable resp.). Let $\mathcal{P}[x_{i_1} = a_{i_1}][x_{i_2} = a_{i_2}] \ldots [x_{i_m} = a_{i_m}]$ be the *sub-problem* obtained from $\mathcal{P}$ by assigning value a_{i_1} to variable x_{i_i}, assigning value a_{i_2} to variable x_{i_2},. . . , assigning value a_{i_m} to variable x_{i_m}.

We reuse the definition of firstx($\mathcal{P}$) defined in QCSPs. The *A-cost* of a QWCSP $\mathcal{P}$, denoted by $\text{A-cost}(\mathcal{P})$, is defined recursively as follows:

$$\text{A-cost}(\mathcal{P}) = \begin{cases} cost(l), \text{ if } \text{firstx}(\mathcal{P}) = \perp \\ \max(\mathbb{M}_i), \text{if } \text{firstx}(\mathcal{P}) = x_i \text{ and } Q_i = \max \\ \min(\mathbb{M}_i), \text{if } \text{firstx}(\mathcal{P}) = x_i \text{ and } Q_i = \min \end{cases}$$

where l is the complete assignment of the completely assigned problem $\mathcal{P}$ (i.e. $\text{firstx}(\mathcal{P}) = \perp$), and $\mathbb{M}_i$ = $\{\text{A-cost}(\mathcal{P}[x_i = v])|v \in D_i\}$.

A QWCSP $\mathcal{P}$ is *satisfiable* iff $\text{A-cost}(\mathcal{P}) < k$. Similar to QCSPs, we define a *block* of variables in a QWCSP $\mathcal{P}$ to be a maximal subsequence of variables in $\mathcal{X}$ which has the same quantifiers. Changing the variable ordering within the same block of variables does not change the A-cost of a QWCSP.

For a maximizatin variable, the goal is to maximize costs. The global upper bound k in QWCSPs means the maximum destructor for maximum variables. Once a maximization variable encounters an A-cost k in a sub-problem, k must be the resulting A-cost. Similarly, 0 means the minimum possible costs for minimization.

Example 12. *Given a QWCSP $\mathcal{P}$ with the ordered sequence of variables (x_1,x_2,x_3), domains $D_1 = \{a, b, c\}, D_2 = \{a, b\}$, and $D_3 = \{a, b, c\}$, the set of constraints represented in Figure 3.1, the quantifier sequence $(Q_1 = \max, Q_2 = \min, Q_3 = \max)$, and the global upper bound k. The problem is to find the A-cost of $\mathcal{P}$. Figure 3.1 indicates there are 3 unary constraints C_1, C_2, C_3 and 2 binary constraints $C_{1,2}, C_{2,3}$. For unary constraints, non-zero unary costs are depicted inside a circle and domain values are placed above the circle. For binary constraints, non-zero binary costs are depicted as labels on edges connecting the corresponding pair of values. Only non-zero costs are shown. We show the computation for the* A-cost *of the QWCSP $\mathcal{P}$ as follows:*

$$\text{A-cost}(\mathcal{P}) = \max_{v_1 \in D_1}\{\min_{v_2 \in D_2}\{\max_{v_3 \in D_3}\{\text{A-cost}(\mathcal{P}[x_1 = v_1][x_2 = v_2][x_3 = v_3])\}\}\}$$

$$
\begin{aligned}
= \quad & \max\{\ \min\{\max\{cost(a,a,a), cost(a,a,b), cost(a,a,c)\}, \max\{cost(a,b,a), cost(a,b,b), cost(a,b,c)\}\},\\
& \min\{\max\{cost(b,a,a), cost(b,a,b), cost(b,a,c)\}, \max\{cost(b,b,a), cost(b,b,b), cost(b,b,c)\}\},\\
& \min\{\max\{cost(c,a,a), cost(c,a,b), cost(c,a,c)\}, \max\{cost(c,b,a), cost(c,b,b), cost(c,b,c)\}\}\ \}\\
= \quad & \max\{\ \min\{\max\{10,5,4\}, \max\{11,8,6\}\},\\
& \min\{\max\{7,2,1\}, \max\{7,4,2\}\},\\
& \min\{\max\{6,1,0\}, \max\{8,5,3\}\}\ \}\\
= \quad & \max\{\ \min\{10,11\}, \min\{7,7\}, \min\{6,8\}\ \}\\
= \quad & \max\{\ 10,7,6\ \} = 10
\end{aligned}
$$

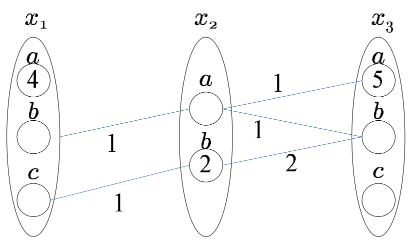

Figure 3.1: Constraints for Example 1

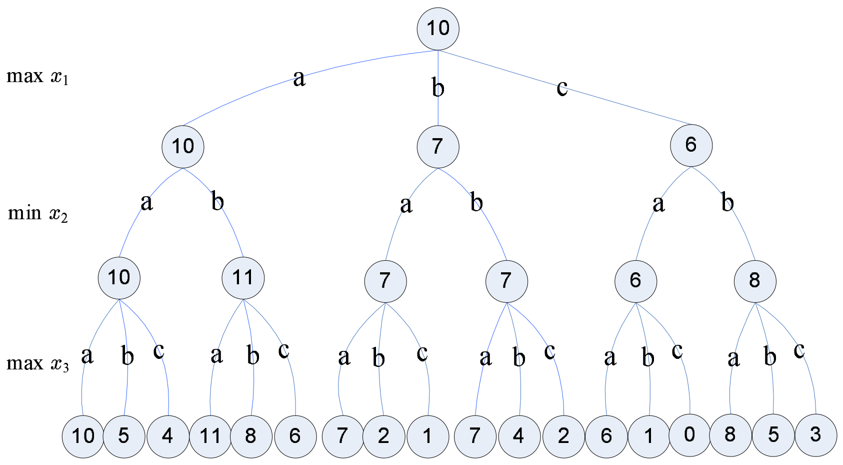

Figure 3.2: Labeling Tree for Example 1

If $k > 10$ in Example 12, then the problem is satisfiable. Otherwise, Example 12 is unsatisfiable. Solution of a WCSP is a complete assignment with the minimum costs, while solution in QCSPs is winning strategy [11]. We define a *solution* of a QWCSP $\mathcal{P}$ as a complete assignment $\{x_1 = v_1, \ldots, x_n = v_n\}$ s.t.: $\text{A-cost}(\mathcal{P}) = \text{A-cost}(\mathcal{P}[x_1 = v_1] \ldots [x_i = v_i]), \forall 1 \leq i \leq n$. Extracting solutions is easy after computing the A-cost of a QWCSP. The complete assignment $\{x_1 = a, x_2 = a, x_3 = a\}$ is a solution of Example 12.

We can use Min-Max trees based on labeling trees [1] for CSPs to explain the semantics and solution space of QWCSPs. The root node of the tree is defined as level 1, and the level of a node is equal to the level of its parents plus one. Nodes at level $i \in [1..n]$ are labelled as ‘Max’ nodes (‘Min’ nodes resp.) if $Q_i = \max$ ($Q_i = \min$ resp.). Suppose $Q_1 = \max$. The A-cost of $\mathcal{P}$ is equal to the maximum A-cost of all sub-problems $\mathcal{P}[x_1 = v_1]$ where $v_1 \in D_1$. We label the root node as a ‘Max’ node, indicating we are choosing the maximum costs for all sub-problems $\mathcal{P}[x_1 = v_1]$. On the other hand if $Q_1 = \min$. The A-cost of $\mathcal{P}$ is equal to the minimum A-cost of all sub-problems $\mathcal{P}[x_1 = v_1]$ where $v_1 \in D_1$. We label the root node as a ‘Min’ node, indicating we are choosing the minimum costs for all sub-problems $\mathcal{P}[x_1 = v_1]$. Followed by the same reasonings, we can infer labelings for all the child nodes inductively. Leaf nodes in the tree represent complete assigned sub-problems, and we label the leaf nodes by their costs. Figure 3.2 shows the Min-Max tree for Example 12. The A-cost for each sub-problem is placed inside the corresponding node. We can easily infer the A-cost of the QWCSP by viewing its Min-Max tree from bottom to top.

Theorem 3.1. *A WCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$ can be transformed by Karp reduction [2] to an equivalent QWCSP $\mathcal{P}' = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}, k)$ with all $Q_i \in \mathcal{Q}$ equal to the* $\min$ *quantifier.*

Proof. As all quantifiers in $\mathcal{P}'$ are $\min$ quantifiers, finding the A-cost of $\mathcal{P}'$ is equivalent in finding the minimum costs among all feasible complete assignments. If $\text{A-cost}(\mathcal{P}') < k$, then the solution of $\mathcal{P}'$ is a complete assignment giving the minimum costs, and it must be a solution of $\mathcal{P}$. If $\text{A-cost}(\mathcal{P}') \geq k$, then the minimum costs of $\mathcal{P}$ must be greater than or equal to k, and $\mathcal{P}$ does not have any solutions. $\square$

Given a hard constraint C. We can construct a soft constraint C' on the same set of variables. A soft constraint C' returns cost 0 if C is satisfiable on the same set of assignments; otherwise, C' returns cost k.

Theorem 3.2. *A QCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q})$ can be transformed by Karp reduction [2] to an equivalent QWCSP $\mathcal{P}' = (\mathcal{X}, \mathcal{D}, \mathcal{C}', \mathcal{Q}', 1)$ where $\mathcal{C}'$ is the set of constraints constructed from $\mathcal{C}$. For each $Q'_i \in \mathcal{Q}'$, if $x_i \in \mathcal{X}$ is an existential variable in QCSP, then Q'_i is a $\min$ quantifier; otherwise, Q'_i is a $\max$ quantifier.*

Given a QCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q})$, we construct a QWCSP $\mathcal{P}' = (\mathcal{X}, \mathcal{D}, \mathcal{C}', \mathcal{Q}', 1)$. For each constraint $C \in \mathcal{C}$, we construct a constraint $C' \in \mathcal{C}'$ on the same set of variables. C' returns cost 0 if C is satisfiable on the same set of assignments. Otherwise, C' returns cost 1.

- Base case: $\text{firstx}(\mathcal{P})$ = $\perp$

  All variables are assigned in this case. Let the complete assignment be l. If all constraints $C \in \mathcal{C}$ are satisfied, i.e. $\mathcal{P}$ is satisfiable, then all constraints $C' \in \mathcal{C}'$ must return costs 0. This implies $cost(l) = 0$ and further implies $\text{A-cost}(\mathcal{P}') = 0$. If at least one constraint $C \in \mathcal{C}$ are not satisfied, i.e. $\mathcal{P}$ is not satisfiable, then at least one constraint $C' \in \mathcal{C}'$ returns cost 1. This implies $cost(l) = 1$ and further implies $\text{A-cost}(\mathcal{P}') = 1$.

- firstx($\mathcal{P}$) = x_i and $Q_i = \exists$

  Assume the simplified sub-problem $\mathcal{P}[x_i = a]$ is satisfiable iff $\text{A-cost}(\mathcal{P}'[x_i = a]) = 0$. By definition of QCSP, we can easily observe:

$$\mathcal{P} \text{ is satisfiable} \iff \exists a \in D_i \;\; s.t. \;\; \mathcal{P}[x_i = a] \text{ is satisfiable}$$

  By the assumption, we can see:

$$\iff \exists a \in D_i \;\; s.t. \;\; \text{A-cost}(\mathcal{P}'[x_i = a]) = 0$$

  Since there exists a sub-problem with A-cost 0,

$$\iff \min_{a \in D_i}(\mathcal{P}'[x_i = a]) = 0$$

By definition of QWCSP,

$$\iff \text{A-cost}(\mathcal{P}') = 0$$

- firstx($\mathcal{P}$) = x_i and $Q_i = \forall$

  Assume the simplified sub-problem $\mathcal{P}[x_i = a]$ is satisfiable iff $\text{A-cost}(\mathcal{P}'[x_i = a]) = 0$. By definition of QCSP,

$$\mathcal{P} \text{ is satisfiable} \quad \iff \forall a \in D_i \;\; s.t. \;\; \mathcal{P}[x_i = a] \text{ is satisfiable}$$

  By assumption,

$$\iff \forall a \in D_i \;\; s.t. \;\; \text{A-cost}(\mathcal{P}'[x_i = a]) = 0$$

  Since all sub-problems with A-cost 0,

$$\iff \max_{a \in D_i}(\mathcal{P}'[x_i = a]) = 0$$

  By definition of QWCSP,

$$\iff \text{A-cost}(\mathcal{P}') = 0$$

Corollary 3.3. *QCSPs and WCSPs are special cases of QWCSPs, which are PSPACE-hard.*

Chapter 4

Branch & Bound with Consistency Techniques

This chapter outlines a complete solver for QWCSPs. The key idea of the solver is that, by applying alpha-beta pruning [38] in the Branch & Bound search and adapting consistency techniques used in WCSPs [23], we can estimate the $\mathrm{A\text{-}cost}$ early in the search so as to reduce the search space.

We first discuss alpha-beta pruning. Then, we describe how various consistency notions, (e.g. NC*, AC*) used in WCSPs, can be modified and integrated with alpha-beta pruning to solve QWCSPs more efficiently.

4.1 Alpha-Beta Pruning

Alpha-beta pruning attempts to reduce search nodes in a minimax algorithm by exploiting (a) semantics of the $\max$ and $\min$ quantifiers and (b) the upper and lower bounds of the costs of previously visited nodes. We can apply alpha-beta pruning in the Branch & Bound search directly in solving QWCSPs as only $\max$ and $\min$ quantifiers are allowed. Alpha-beta pruning is also used for solving real-time online QCSPs [41].

Figure 4.2 is a high-level abstraction of a QWCSP solver. We first neglect the propagation routine (the grey box) in lines 5–18 and discuss the basic alpha-beta

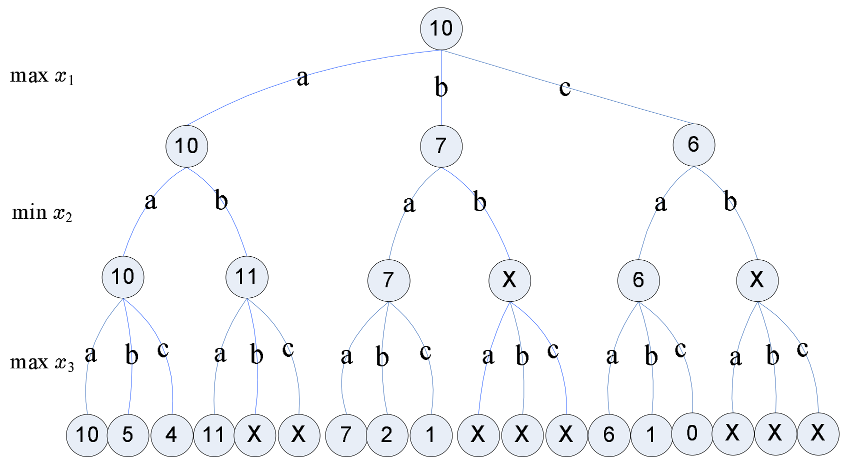
max x_1
min x_2
max x_3
10
a
b
c
10
7
6
a b a b a b
10 11 7 X 6 X
a b c a b c a b c a b c a b c a b c
10 5 4 11 X X 7 2 1 X X X 6 1 0 X X X

Figure 4.1: Labeling Tree for Example 1 after applying alpha-beta pruning

pruning algorithm. The search starts with `alpha_beta(P,0,k)`. We denote the input QWCSP $\mathcal{P}$ by `P` in Figure 4.2. The bounds lb and ub are the range of costs found by the alpha-beta pruning algorithm. QWCSP propagators further exploit these bounds to achieve stronger propagation. We use `P[xj != u]` to denote a function pruning value u of variable x_j in $\mathcal{P}$, and `P[xi = v]` to denote a function assigning value v to variable x_i in $\mathcal{P}$. Both functions return the modified problem. Line 2 is the base case in which all variables are bound. The routine `cost` returns the cost of the complete assignment. Lines 19–24 give the main routine of the traditional alpha-beta pruning algorithm. We only explain the cost for the $\min$ quantifier, since that of $\max$ is similar. The for loop in lines 4–24 evaluates all sub-problems $\mathcal{P}[x_i = v]$ by recursively invoking the alpha-beta algorithm. Since the goal is to find a minimum value, the upper bound is updated. When the upper bound is less than the lower bound (line 24), it triggers the short-cut to break out of the remaining search since every value returned by subsequent calls will be dominated by the current bounds. The function `alpha_beta` ends by returning the upper bound for the $\min$ quantifier (line 25). We illustrate the code with an example.

Example 13. *Consider the tree in Figure 3.2, and assume values are labeled in the sequence of* $[a, b, c]$*. The search starts with* `alpha_beta(P,0,k)`*. Consider the node* $\mathcal{P}' = \mathcal{P}[x_1 = a]$*, which first visits its sub-problem* $\mathcal{P}'[x_2 = a]$ *by calling*

```
1  function alpha_beta(P,lb,ub):
2    if firstx(P) == bot: return cost(P)
3    xi = firstx(P)
4    for v in Di:
5      // Pruning using QWCSP semantics
6      changed = true
7      while changed:
8      changed = false
9        for j in i..n:
10         for u in Dj:
11           ap_lb = approx_lb(P, xj = u)
12           if ub <= ap_lb:
13             if Qj == min: P = P[xj != u], changed = true
14             else: return ub
15           ap_ub = approx_ub(P, xj = u)
16           if ap_ub <= lb:
17             if Qj == min: return lb
18             else: P = P[xj != u], changed = true
19     // Basic alpha-beta pruning
20     if Qi == min:
21       ub = min(ub, alpha_beta(P[xi = v],lb,ub))
22     else:
23       lb = max(lb, alpha_beta(P[xi = v],lb,ub))
24     if ub <= lb: break
25   return (Qi == min)?ub:lb
```

Figure 4.2: A QWCSP Solver

`alpha_beta(P'`$[x_2 = a]$`,0,k)` and a value of 10 is returned. Since $Q_2 = \min$*, the upper bound is updated. The routine then invoke `alpha_beta(P'`$[x_2 = b]$`,0,10)`. After visiting* $\mathcal{P}'[x_2 = b][x_3 = a]$*, we get an* A-cost *of 11 for the sub-problem. The quantifier here is* $\max$*, the cost is greater than the upper bound and the condition in line 24 holds. No matter what costs the remaining sub-problems produce, they have no impact on the solution and a short-cut to break out of the remaining search is triggered. Hence, the sub-problems* $\mathcal{P}'[x_2 = b][x_3 = b]$ *and* $\mathcal{P}'[x_2 = b][x_3 = c]$ *are not explored.*

Figure 4.1 illustrates the nodes pruned, denoted by the symbol X, by alpha-beta pruning.

4.2 Consistency Techniques

In traditional CSPs, we enforce different levels of consistency to prune infeasible domain values and hence reduce the search space. In WCSPs, the consistency algorithms take the cost of constraints into account. Various consistency notions (e.g. NC*, AC* [23], FDAC*, EDAC*, OSAC, and VAC [17]) have been proposed and proven to be useful in improving solver performance. Such techniques, however, cannot be directly applied to QWCSP since the quantifiers change the semantics of constraints.

To prune values of a QWCSP, the main idea is that if the $\mathrm{A\text{-}cost}$ of a sub-problem $\mathcal{P}' = \mathcal{P}[x_i = v]$ is greater than or equal to the upper bound ub (less than or equal to the lower bound lb resp.), the pruning techniques in alpha-beta can be applied. Let $\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]$ denote the subproblem $\mathcal{P}[x_1 = v_1][x_2 = v_2]...[x_{i-1} = v_{i-1}][x_i = v]$. Formally, we consider two conditions: $\exists v \in D_i \quad s.t. \quad \forall v_1 \in D_1, ..., v_{i-1} \in D_{i-1}$:

$$\text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \geq ub \tag{4.1}$$

$$\text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \leq lb \tag{4.2}$$

When any of the above conditions is satisfied, we can apply alpha-beta pruning according to Table 4.1.

A-cost	$\geq ub$	$\leq lb$
$Q_i = \min$	prune v	backtrack
$Q_i = \max$	backtrack	prune v

Table 4.1: When can we prune/backtrack

Theorem 4.1. *Given a QWCSP $\mathcal{P}$. If Condition (1)/(2) for $\mathcal{P}$ is satisfied, applying prunings and backtrackings according to Table 4.1 is sound.*

Proof. (Sketch) Reasons to perform prunings and backtracking for $\min$ and $\max$ are symmetrical. We only describe the case where $Q_i = \min$. Suppose Condition (1) holds. We consider A-cost(s) for sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}]$. Without loss of generality, we write $\mathcal{P}_{i-1}$ to be one of these sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}]$ by fixing values $v_1 \in D_1$, $v_2 \in D_2$, $\ldots$, $v_{i-1} \in D_{i-1}$. We will see the proof using $\mathcal{P}_{i-1}$ applies for all sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}]$, regardless on which values we fix. Given $Q_i = \min$, we obtain:

$$\text{A-cost}(\mathcal{P}_{i-1}) = \min_{a \in D_i} \mathcal{P}_{i-1}[x_i = a]$$

If $\text{A-cost}(\mathcal{P}_{i-1}) < ub$, the following must be true:

$$\exists v' \in D_i \text{ where } v' \neq v \;\; s.t. \;\; \text{A-cost}(\mathcal{P}_{i-1}[x_i = v']) < ub$$

Pruning value v does not change the A-cost of P_{i-1}. If $\text{A-cost}(\mathcal{P}_{i-1}) \geq ub$, i.e. $\mathcal{P}_{i-1}$ must not lead to solutions, the following must be true:

$$\forall v' \in D_i, \text{A-cost}(\mathcal{P}_{i-1}[x_i = v']) \geq ub$$

After pruning value v, either domain wipe out occurs or $\text{A-cost}(\mathcal{P}_{i-1}) \geq ub$. For both cases, the sub-problem $\mathcal{P}_{i-1}$ cannot lead to solutions. Combining the two cases, pruning value v does not change the problem from unsatisfiable to satisfiable(,and vice versa).

We now discuss Condition (2). Similar to the previous case, we consider the A-cost for these sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}]$, and we fix $\mathcal{P}_{i-1}$ to be one of these sub-problems similarly. Given $Q_i = \min$, we obtain:

$$\text{A-cost}(\mathcal{P}_{i-1}) = \min_{a \in D_i} \mathcal{P}_{i-1}[x_i = a]$$

By Condition (2), $\mathcal{P}_{i-1}[x_i = v] \leq lb$ holds, and therefore:

$$\text{A-cost}(\mathcal{P}_{i-1}) \leq lb$$

Recall $\text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \leq lb$ applies regardless on which value v_1, v_2, $\ldots$, v_{i-1} we fix. Therefore, we obtain:

$$\forall v_1 \in D_1, ..., v_{i-1} \in D_{i-1}, \text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}]) \leq lb$$

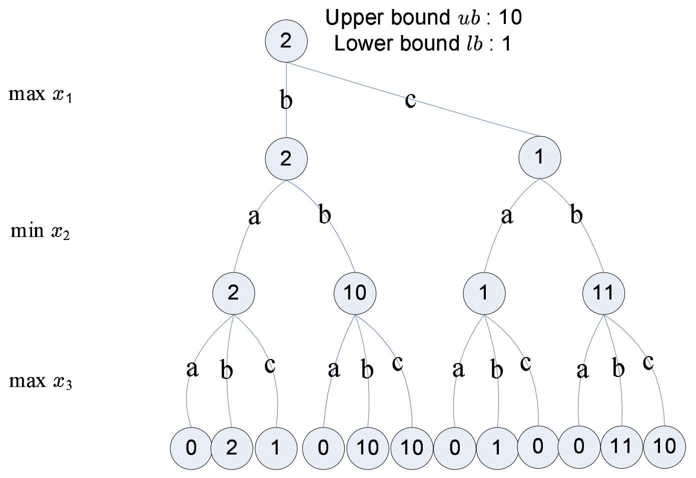

Figure 4.3: Labeling Tree for Example 14

We can easily obtain the following results using the definition of A-cost for a QWCSP:

$$\text{A-cost}(\mathcal{P}) \leq lb$$

QWCSP $\mathcal{P}$ must be unsatisfiable, and the solver can backtrack. □

Example 14. *Given a QWCSP $\mathcal{P}$ with the ordered sequence of variables* (x_1,x_2,x_3), *domains* $D_1 = \{a, b, c\}, D_2 = \{a, b\}$, *and* $D_3 = \{a, b, c\}$, *the quantifier sequence* $(Q_1 = \max, Q_2 = \min, Q_3 = \max)$, *and the global upper bound* 10. *Suppose the* A-cost *for the sub-problem* $\mathcal{P}[x_1 = a]$ *is 1. Figure 4.3 shows the upper bound* ub, *lower bound* lb, *and the* A-cost *for the remaining sub-problem. By inspecting the figure, we can easily observe Condition (2) holds:* $\exists a \in D_3 \;\; s.t. \;\; \forall v_1 \in D_1, \forall v_2 \in D_2$,

$$\text{A-cost}(P[x_1 = v_1][x_2 = v_2][x_3 = a]) \leq lb$$

By Table 4.1, we can prune value a of x_3. We can easily observe the solution must not contain the assignment $x_3 = a$, and therefore, we prune the value. After pruning value a of x_3, we can easily observe Condition (1) holds: $\exists b \in D_2 \;\; s.t. \;\; \forall v_1 \in D_1$,

$$\text{A-cost}(P[x_1 = v_1][x_2 = b]) \geq ub$$

By similar reasons, we can prune value b of x_2.

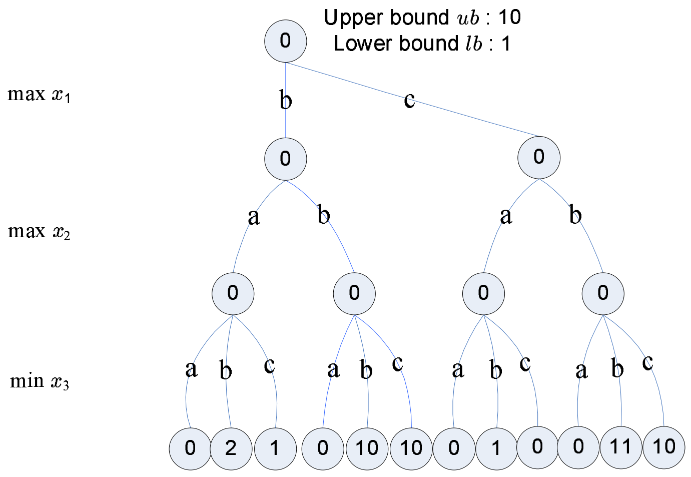

Figure 4.4: Labeling Tree for Example 15

Example 15. *Suppose the quantifier sequence of Example 14 is replaced by* $(Q_1 = \max, Q_2 = \max, Q_3 = \min)$*, and the* A-cost *for the sub-problem* $\mathcal{P}[x_1 = a]$ *remain unchanged* ($\text{A-cost}(\mathcal{P}[x_1 = a]) = 1$)*. Figure 4.4 shows the upper bound* ub*, lower bound* lb*, and the* A-cost *for the remaining sub-problem. Costs for each complete assignment remain the same as in Example 14. The only difference is the modified* A-cost *for sub-problems, resulting from the change in quantifiers. By inspecting the figure, we can easily observe Condition (2) still holds:* $\exists a \in D_3 \ \ s.t. \ \ \forall v_1 \in D_1, \forall v_2 \in D_2,$

$$\text{A-cost}(\mathcal{P}[x_1 = v_1][x_2 = v_2][x_3 = a]) \leq lb$$

As $Q_3 = \min$*, we can easily observe all the* A-cost *for sub-problems* $\mathcal{P}[x_1 = v_1][x_2 = v_2], \forall v_1 \in D_1, v_2 \in D_2$ *must be less than or equal to the* lb*. By induction, we can conclude sub-problems* $\mathcal{P}[x_1 = v_1], \forall v_1 \in D_1$ *must be less than or equal to the* lb*, and finally obtain* $\text{A-cost}(\mathcal{P}) \leq lb$*. Therefore following Table 4.1, the solver can perform backtrack.*

In other words, Condition (1) and (2) are sufficient conditions for the prunings and backtrackings in Table 4.1.

To check Condition (1)/(2), one way is to find the exact value of A-cost for each sub-problem, which is computationally expensive. The problem is essentially equivalent to determining if a variable assignment belongs to a solution in classical CSPs in general, which is NP-hard. A common technique in constraint programming is to formulate consistency notions and algorithms, which aim at extracting information in a problem to make it explicit. Useful information includes pruning and cost information. Here we apply the same idea. We give some efficient ways to extract a good upper bound and lower bound of A-cost, so as to backtrack or identify non-solution values from domains early in the search.

In the QWCSP solver (Figure 4.2), lines 5–18 prune or backtrack according to the conditions specified in Table 4.1. Since finding A-cost is difficult, the algorithm finds the approximated bounds (`approx_lb` in line 11, and `approx_ub` in line 15). Functions `approx_lb`$(\mathcal{P}, x_i = v)$ and `approx_ub`$(\mathcal{P}, x_i = v)$ find the approximate A-cost for the set S of sub-problems, where:

$$S = \{\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] | \forall v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}\}$$

such that:

$$\forall P' \in S, \text{A-cost}(P') \leq \texttt{approx\_ub}(\mathcal{P}, x_i = v) \text{ ,and}$$
$$\forall P' \in S, \text{A-cost}(P') \geq \texttt{approx\_lb}(\mathcal{P}, x_i = v)$$

`approx_ub`$(\mathcal{P}, x_i = v)$ is *tight* if:

$$\max_{P' \in S} \text{A-cost}(P') = \texttt{approx\_ub}(\mathcal{P}, x_i = v)$$

Similarly, `approx_lb`$(\mathcal{P}, x_i = v)$ is *tight* if:

$$\min_{P' \in S} \text{A-cost}(P') = \texttt{approx\_lb}(\mathcal{P}, x_i = v)$$

Corollary 4.2. *Given a QWCSP $\mathcal{P}$, If `approx_ub`$(\mathcal{P}, x_i = v) \leq lb$, we can prune value v of variable x_i if $Q_i = \max$, and perform backtrack if $Q_i = \min$. If `approx_lb`$(\mathcal{P}, x_i = v) \geq ub$, we can prune value v of variable x_i if $Q_i = \min$, and perform backtrack if $Q_i = \max$.*

Proof. (Sketch) We can easily observe the following:

$$lb \geq \texttt{approx\_ub}(\mathcal{P}, x_i = v) \geq \mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]$$
$$ub \leq \texttt{approx\_lb}(\mathcal{P}, x_i = v) \leq \mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]$$

$\forall v_1 \in D_1, v_2 \in D_2, \ldots, v_{i-1} \in D_{i-1}$. Condition $\texttt{approx\_ub}(\mathcal{P}, x_i = v) \leq lb$ implies Condition (2), and condition $\texttt{approx\_lb}(\mathcal{P}, x_i = v) \geq ub$ implies Condition (1). We then apply Table 4.1 according to these conditions. □

We have designed two kinds of consistency notions: node consistency and arc consistency, which extract the approximated upper and lower bounds (hence implement `approx_ub` and `approx_lb`) for QWCSPs. For simplicity, we restrict our discussion only to zero-arity constraint $C_\varnothing$, unary constraints C_i, and binary constraints C_{ij}.

We first discuss *Node Consistency* notions which use *mainly* unary constraints to perform pruning. Then we discuss *Arc Consistency* which takes all of $C_\varnothing$, unary constraints and binary constraints into account. Our algorithms can be generalized to higher-arity constraints.

4.2.1 Node Consistency

Overview

The Node Consistency algorithm depends on three components: a lower bound for $\mathrm{A\text{-}cost}$, similarly an upper bound, and a weak-NC* notion. We first discuss the intuition and definition for each of them, and then demonstrate how to integrate them with our proposed QWCSP solver (Figure 4.2).

Lower Bound of A-Cost

We consider computing bounds by closely examining unary constraints. The function $nc_{lb}(\mathcal{P}, x_i = v)$ returns a lower bound for all sub-problems with value v assigned to variable x_i.

Definition 4.3. *The function $nc_{lb}(\mathcal{P}, x_i = v)$ approximates the* A-cost *for a set S of sub-problems:*

$$\{\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] | \forall v_1 \in D_1, v_2 \in D_2, \ldots, v_{i-1} \in D_{i-1}\}$$

in which value v is assigned to variable x_i:

$$nc_{lb}(\mathcal{P}, x_i = v) \equiv C_\varnothing \oplus C_i(v) \oplus \bigoplus_{j:i<j \wedge Q_j = \min} \min_{u \in D_j} C_j(u) \bigoplus_{j:i<j \wedge Q_j = \max} \max_{u \in D_j} C_j(u)$$

where Q_j is the quantifier for variable x_j.

Theorem 4.4. $\forall v_1 \in D_1, ..., v_{i-1} \in D_{i-1}$, A-cost$(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \geq nc_{lb}(\mathcal{P}, x_i = v)$.

By Theorem 4.4, the function $nc_{lb}(\mathcal{P}, x_i = v)$ returns a lower bound for all sub-problems with value v assigned to x_i.

Proof. (sketch) $C_\varnothing$ gives the global minimum costs, and $C_i(v)$ gives costs when value v is assigned to variable x_i. Including these two terms to the lower bound estimation is trivial as these costs must incurred to all sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]$. Intuitively, we should count the minimum unary costs for variables x_j where $j < i$ to make the lower bound tighter. We choose to ignore these unary costs. The reason behind is that we will perform unary projections projecting the minimum costs of each unary constraint to $C_\varnothing$, and as a result, the minimum costs for these unary constraints are always 0. Consider the node corresponding to sub-problem $\mathcal{P}$ with $x_i = v$. If $Q_{i+1} = \max$, the cost of its child node is at least the maximum cost of C_{i+1}, since the $\max$ quantifier will choose a solution at least as great as $\max C_{i+1}$. Likewise, if $Q_{i+1} = \min$, the $\min$ quantifier will choose sub-problem with the minimum cost. The reasoning process continues recursively until the leaf node. Therefore, we can add unary costs given by $C_j, j > i$ according to the quantifier Q_j to make the lower bound estimation tighter. $\square$

To simplify our notations, we write the minimum costs $\min_{u \in D_j} C_j(u)$ and maximum costs $\max_{u \in D_j} C_j(u)$ of a unary constraint C_j as follows:

$$\min C_j = \min_{u \in D_j} C_j(u)$$

$$\max C_j = \max_{u \in D_j} C_j(u)$$

We then define $Q_j C_j$ as follows:

$$Q_j C_j = \begin{cases} \min C_j & \text{if } Q_j = \min \\ \max C_j & \text{if } Q_j = \max \end{cases}$$

We can now write function $nc_{lb}(\mathcal{P}, x_i = v)$ in a more compact way.

$$nc_{lb}(\mathcal{P}, x_i = v) \equiv C_\varnothing \oplus C_i(v) \oplus \bigoplus_{j:i<j} Q_j C_j$$

Readers may be aware that there may be ways to achieve a tighter bound: 1) replace $i < j$ by $i \neq j$, and 2) modify $nc_{lb}(\mathcal{P}, x_i = v)$ such that it returns a lower bound of the A-cost of $P[x_i = v]$. However, these changes lead to unsound prunings, which we demonstrate in Example 16.

Example 16. *Given a QWCSP $\mathcal{P}$ with the ordered sequence of two variables (x_1, x_2), domains $D_1 = D_2 = \{a, b\}$, two unary constraints C_1 and C_2, one binary constraint $C_{1,2}$, and a quantifier sequence $(Q_1 = \max, Q_2 = \min)$. We denote the global upper bound by k. Non-zero costs given by constraints are listed as follows: $C_1(a) = 50$, $C_2(a) = 9$, $C_{1,2}(b, b) = k$. If $k = 59$, the A-cost of $\mathcal{P}$ is 50. Figure 4.5 shows the labeling tree for $\mathcal{P}$. If $i < j$ is replaced by $i \neq j$ in Definition 4.3, $nc_{lb}(\mathcal{P}, x_2 = a)$ returns 59 $>$ $\text{A-cost}(\mathcal{P}[x_1 = b][x_2 = a])$ = 9. If $nc_{lb}(\mathcal{P}, x_2 = a)$ returns the lower bound approximation of $\text{A-cost}(\mathcal{P}[x_2 = a])$, it may return 59. Both cases may cause value a of x_2 to be pruned. Figure 4.6 shows the labeling tree after pruning, and we can easily observe the A-cost of the new problem changes from 50 to 59. The pruning is unsound.*

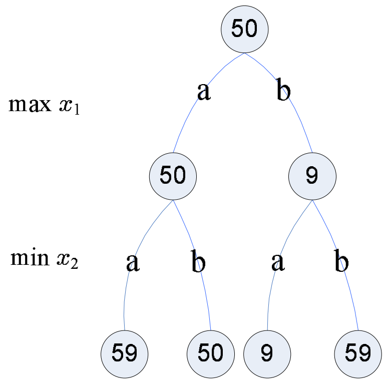

Figure 4.5: Labeling Tree for Example 16

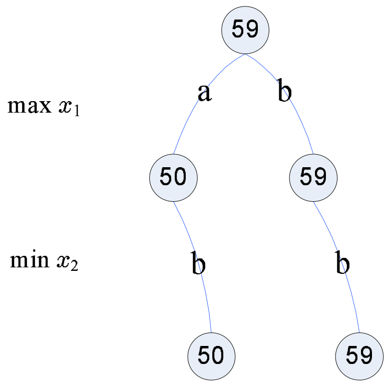

Figure 4.6: Labeling Tree for Example 16 after pruning a of x_2

Upper Bound of A-Cost

Computing the upper bound is less straightforward. Unlike its lower bound counterpart, in which we know the minimum cost is always 0, we need to take *all* constraints into account. In particular, maximum costs $\max_{u \in D_i, v \in D_j} C_{ij}(u, v)$ for each binary constraint C_{ij} are needed in order to compute a correct upper bound. We use $\max C_{ij}$ to denote the maximum costs $\max_{u \in D_i, v \in D_j} C_{ij}(u, v)$ for a binary constraint C_{ij}.

Definition 4.5. *The function nc_{ub} approximates the* A-cost *for a set S of sub-problems: $\{P[x_{1..i-1} = v_{1..i-1}, x_i = v] | \forall v_1 \in D_1, v_2 \in D_2, \ldots, v_{i-1} \in D_{i-1}\}$ in which value v is assigned to variable x_i:*

$$nc_{ub}(\mathcal{P}, x_i = v) \equiv C_\varnothing \oplus C_i(v) \oplus \bigoplus_{i<j} Q_j C_j \oplus \bigoplus_{j<i} \max C_j \oplus \bigoplus_{j \neq k} \max C_{jk}$$

Theorem 4.6. $\forall v_1 \in D_1, ..., v_{i-1} \in D_{i-1}$, A-cost$(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \leq nc_{ub}(\mathcal{P}, x_i = v)$.

By Theorem 4.6, the function $nc_{ub}(\mathcal{P}, x_i = v)$ returns an upper bound for all sub-problems with value v assigned to x_i.

Proof. (sketch) The main idea for the first three terms is similar to that of the lower bound. However, to get the upper bound, we have to take into account of every constraint in the problem. Otherwise, the upper bound may be incorrect. We consider the maximum unary costs, for variables x_j where $j < i$. Maximum costs given by binary constraints, which are precomputed before search, are used as a weak upper bound for binary constraints. The fourth term takes the maximum cost for unary constraints before i, and the fifth considers all binary constraints. Recall that, for simplicity, we only allow $C_\emptyset$, unary constraints, and binary constraints. $\square$

Similar to nc_{lb}, there may be ways to achieve a tighter bound for nc_{ub}: 1) replace $i < j$ by $i \neq j$, and 2) modify $nc_{ub}(\mathcal{P}, x_i = v)$ returning the upper bound for $\mathcal{P}[x_i = v]$. However, due to similar reasons as illustrated in Example 16. These modifications are unsound.

Projecting Unary Costs to $C_\varnothing$

In WCSPs, costs can be projected from higher arity constraint to lower arity ones to achieve propagation. The function nc_{lb} does not take *all* unary constraints into account. If we are able to project some unary costs to $C_\varnothing$, we can make $nc_{lb}(\mathcal{P}, x_i = v)$ a tighter bound.

Suppose there is a variable $x \in D = \{a, b\}$ and a unary constraint C over x where $C(a) = 1$ and $C(b) = 2$. In any assignment, the cost incurred by C is at least 1. We can *project* a cost of 1 from C to the zero-arity constraint $C_\varnothing$. After projection, the cost of C is therefore changed to $C(a) = 0$ and $C(b) = 1$ and $C_\varnothing = 1$. Projection *preserves problem equivalence*, and it is discussed in the background section. Before computing the bounds, we require *weak-NC** to hold to achieve stronger propagation.

Definition 4.7 (Weak-NC*)**.** $\forall i, \exists v \in D_i : C_i(v) = 0.$

By enforcing weak-NC* using unary projections, we can make $nc_{lb}(\mathcal{P}, x_i = v)$ a tighter bound. However, weak-NC* does not improve nc_{ub} since the maximum cost of all constraints have already been taken into account. For completely instantiated unary constraints, weak-NC* will project costs w.r.t. the assigned value to $C_\varnothing$.

Example 17 (Lower Bound)**.** *We re-use Example 13. Suppose we are at sub-problem $\mathcal{P}' = \mathcal{P}[x_1 = a]$ and we have just visited the further sub-problem $\mathcal{P}'[x_2 = a]$ which have a new upper bound of 10. Before visiting $\mathcal{P}'[x_2 = b]$, we try to prune some values according to Table 4.1 using the new upper bound. In particular, nc_{lb} is applied. First, since x_1 is bound, the unary cost $C_1(a)$ is projected to $C_\varnothing$ and we* *have $C_\varnothing$ = 4. We want to check if the value b can be pruned from D_2. In the sub-problem $\mathcal{P}'[x_2 = b]$, the quantifier Q_3 is $\max$, and it will take at least the maximum unary cost $\max C_3$. We have $C_\varnothing + C_2(b) + \max C_3 = 4 + 2 + 5 \geq 11 > ub$. The cost of any assignment in the sub-problem $\mathcal{P}'[x_2 = b]$ is at least 11, which is greater than the upper bound 10. The value b can therefore be removed from domain D_2. Notice that such node is not pruned by basic alpha-beta pruning.*

Example 18 (Upper Bound)**.** *Consider the sub-problem $\mathcal{P}' = \mathcal{P}[x_1 = b][x_2 = a]$. The search algorithm currently has a lower bound of 10. We want to check the* $\text{A-cost}(\mathcal{P}')$*. $C_\varnothing$ has a value of 0, and both C_1 and C_2 incur a cost of zero. It remains to consider all other constraints. The easiest way is to sum up the maximum cost of every remaining constraint. The sum is 8, which is less than the lower bound. According to Table 4.1, the algorithm can backtrack.*

We are now ready to integrate all components to define Node Consistency.

Definition 4.8. *A QWCSP $\mathcal{P}$ is* Node Consistent *(NC) when the following conditions hold:*

$$\forall i, \exists v \in D_i : \; C_i(v) = 0 \quad \textit{(Weak-NC*)}$$

$$\forall x_i \in X, \forall v \in D_i : nc_{lb}(\mathcal{P}, x_i = v) < ub \; \wedge nc_{ub}(\mathcal{P}, x_i = v) > lb$$

where ub and lb are the upper and lower bounds in alpha-beta pruning.

4.2.2 Enforcing Algorithm for NC

Achieving NC using the QWCSP solver (Figure 4.2) is easy. The solver would maintain the three NC conditions: 1) weak-NC*, 2) $nc_{lb}(\mathcal{P}, x_i = v) < ub$, and 3) $nc_{ub}(\mathcal{P}, x_i = v) > lb$, in a step-wise manner. We use projection algorithms from WCSP solvers to maintain weak-NC*. To maintain condition 2) and condition 3), we use pruning/filtering algorithms to prune values v of variables x_i which do not satisfy either condition 2) or condition 3). We then add extra routines for these algorithms to handle backtracking cases. Writing such pruning/filtering algorithms can be seen as implementing functions `approx_lb` (line 11) and `approx_ub` (line 15) in Figure 4.2 by nc_{lb} and nc_{ub} respectively. We note that after enforcing condition 3) and a value v of variable x_i is being pruned, variable x_i may loss weak-NC* property. One way to re-enforce weak-NC* is to allow the propagation routine repeats (the while loop in lines 7 to 19) until no domains are further reduced.

We now explain the NC enforcing algorithm in details, by showing the algorithm `QWNC-1(P)` in Figure 4.7. The algorithm is divided into two phases: projection phase for enforcing weak-NC* (line 6 to 7), and pruning phase (line 9 to 25) for enforcing condition 2) and 3).

Projection Phase

In the projection phase, we perform unary projections for all unary constraints similar to the NC* enforcing algorithm for WCSPs. We re-use function `unaryProject` in the NC* enforcing algorithm (line 7).

Pruning Phase

We then use two functions `QWNC-1-LbApprox` and `QWNC-1-UbApprox` to enforce the two conditions $nc_{lb}(\mathcal{P}, x_i = v) < ub$ and $nc_{ub}(\mathcal{P}, x_i = v) > lb$ respectively in the pruning phase. The main goal of these two functions is to compute expression nc_{lb}/nc_{ub} by gathering required costs from constraints. After the expressions are computed, bounds will then be checked and prunings/backtrackings will be performed accordingly.

Intuitively, we need to form the two expressions $O(nd)$ times (, where n is the number of variables and d is the maximum domain size), as values of all variables are required to be checked. Inspecting costs for unary constraints and binary constraints, which are stored in tables, takes time. One way to reduce the complexity of this procedure is to pre-compute terms in expression nc_{lb} and nc_{ub} before calling the two functions. Terms such as: $\max C_i$, $\bigoplus_{i<j} Q_j C_j$, and $\bigoplus_{j<i} \max C_j$ in nc_{lb}

```
function QWNC-1(P)
  change = true
  while change:
    change = false
    // Projection Phase
    for i in [1..n]:
      unaryProject(Ci)
    // Pruning Phase
    for i in [1..n]:
      MaxUnaryCost[i] = maxCost(Ci)
    for i in [1..n]:
      SumOfUnaryCostsAfter[i] = 0
      for j in [i+1..n]:
        if Qj == max:
          SumOfUnaryCostsAfter[i] += MaxUnaryCost[j]
    for i in [1..n]:
      SumOfUnaryCostsBefore[i] = 0
      for k in [1..i-1]:
        SumOfUnaryCostsBefore[i] += MaxUnaryCost[k]
    if QWNC-1-LbApprox(P,M):
      change = true
      if domainWipeOut(): return
    if QWNC-1-UbApprox(P,M):
      change = true
      if domainWipeOut(): return
  return
function QWNC-1-LbApprox(P,M)
  change = false
  for i in [1..n]:
    for v in Di:
      if C_NULL + C_i(v) + SumOfUnaryCostsAfter[i] >= ub:
        if Qi == min:
          prune(xi,v)
          change = true
        if Qi == max:
          pruneAll(xi)
          change = true
          return change
  return change
function QWNC-1-UbApprox(P,M)
  change = false
  for i in [1..n]:
    for v in Di:
      if C_NULL + C_i(v) + SumOfUnaryCostsAfter[i] +
         SumOfUnaryCostsBefore[i] + binaryMax() <= lb:
        if Qi == max:
          prune(xi,v)
          change = true
        if Qi == min:
          pruneAll(xi)
          change = true
          return change
  return change
```

Figure 4.7: Enforcing algorithm for NC

and nc_{ub} are precomputed before calling the two function `QWNC-1-LbApprox` and `QWNC-1-UbApprox`. We now explained as follows.

To ensure efficient access to the maximum costs $\max C_i$ of unary constraints C_i in both functions, we store these costs in the array `MaxUnaryCost` (line 10), where `MaxUnaryCost[i]` holds the maximum costs for unary constraint C_i. We use the function `maxCost(C)` to compute the maximum possible costs given by a constraint `C`, where the constraint, in general, can be a constraint of any arity.

The two functions also use an array `SumOfUnaryCostsAfter` to compute $\bigoplus_{i<j} Q_j C_j$ for all variables x_i (line 11 to 15). We can view the array element `SumOfUnaryCostsAfter[i]` stores costs generated by unary constraints on variables j where $j > i$.

In addition, `QWNC-1-UbApprox` uses an array `SumOfUnaryCostsBefore` to compute $\bigoplus_{j<i} \max C_j$ for all variables x_i (line 16 to 19). Similar to the array `SumOfUnaryCostsAfter`, we can view the array element `SumOfUnaryCostsBefore[i]` stores costs generated by unary constraints on variables j where $j < i$.

A function `binaryMax()` is used to query the summation of maximum costs for all binary constraints, which is precomputed before search.

We use a function `pruneAll(xi)` to prune all values of variable `xi`, and a function `domainWipeOut()` to detect if there exists a variable with empty domain (i.e. all values being pruned). These two functions are used to detect backtracking cases.

Time Complexity

Theorem 4.9. *`QWNC-1` runs in time $O(n^2d^2)$, where n is the number of variables, and d is the maximum domain size of the corresponding QWCSP.*

Proof. (Sketch)

Time complexity for computing `unaryProject` for all unary constraints and the array `MaxUnaryCost` are both $O(nd)$. To achieve a better time complexity, we choose to pre-compute the two arrays: `SumOfUnaryCostsAfter` and `SumOfUnaryCostsBefore` in a recursive dynamic programming approach before calling `QWNC-1-LbApprox(P,M)` and `QWNC-1-UbApprox(P,M)`. We can express `SumOfUnaryCostsAfter[i]` and `SumOfUnaryCostsBefore[i]` as follows:

$$\texttt{SumOfUnaryCostsAfter}[i] = \begin{cases} 0 & \text{if } i = n \\ \texttt{SumOfUnaryCostsAfter}[i+1] & \\ +\texttt{MaxUnaryCost}[i+1] & \text{if } Q_{i+1} = \max, 1 \le i \le n-1 \\ \texttt{SumOfUnaryCostsAfter}[i+1] & \text{if } Q_{i+1} = \min, 1 \le i \le n-1 \end{cases}$$

$$\texttt{SumOfUnaryCostsBefore}[i] = \begin{cases} 0 & \text{if } i = 1 \\ \texttt{SumOfUnaryCostsBefore}[i-1] & \\ +\texttt{MaxUnaryCost}[i-1] & \text{if } 2 \le i \le n \end{cases}$$

It is easy to see by modifying the computation of array `SumOfUnaryCostsAfter` and `SumOfUnaryCostsBefore` in a recursive dynamic programming approach (line 11 to 15 and line 16 to 19 in Figure 4.7), the computation time reduces to time $O(n)$. `QWNC-1-LbApprox` and `QWNC-1-UbApprox`, both computing costs for each value of all variables, run in time $O(nd)$. If a value is pruned, the propagation routine may repeat. There are $O(nd)$ values, therefore, the propagation routine can repeat at most $O(nd)$ time. Overall, the time complexity for `QWNC-1` is $O(n^2d^2)$. $\square$

4.2.3 Arc Consistency

Overview

Example 19. *Given a QWCSP $\mathcal{P}$ with an ordered sequence of two variables (x_1, x_2), domains $D_1 = D_2 = \{a, b\}$, two unary constraints C_1 and C_2, one binary constraint $C_{1,2}$, a quantifier sequence $(Q_1 = \min, Q_2 = \max)$, and a global upper bound $k = 10$. Non-zero costs given by constraints are listed as follows: $C_1(a)$=3, $C_2(a)$=2, $C_{1,2}(a, b)$=7. Value a of x_1 can be pruned as $\text{A-cost}(\mathcal{P}[x_1 = a]) \geq k$. Enforcing NC on $\mathcal{P}$ cannot prune this value. Even if we apply binary projections on $C_{1,2}$, no costs can be projected to unary constraints to trigger prunings by NC.*

We need to consider quantifier properties in order to define Arc Consistency (AC) in the QWCSP framework. Based on the costs of unary and binary constraints, we can define functions to compute a tighter lower and upper bounds to A-cost. These functions, however, depend on the quantifiers associated with the two variables in the binary constraints under consideration. There are four combinations of $\min$ and $\max$, and two bounds to compute. We have a total of eight functions, split into two categories: functions $ac_{ub}^{Q_i,Q_j}$ for computing the AC upper bound, and functions $ac_{lb}^{Q_i,Q_j}$ for computing the AC lower bound of the A-cost for a set S of sub-problems:

$$\{P[x_{1..i-1} = v_{1..i-1}, x_i = v] | \forall v_1 \in D_1, v_2 \in D_2, \ldots, v_{i-1} \in D_{i-1}\}$$

in which value v is assigned to variable x_i. We always assume $i < j$ in the following definitions.

Lower Bound of A-Cost

Definition 4.10. *The function $ac_{lb}^{Q_i,Q_j}$ approximates the* A-cost *for the set S of sub-problems:*

$$ac_{lb}^{Q_i,Q_j}(\mathcal{P}, x_i = v) \quad \equiv C_\varnothing \oplus C_i(v) \oplus \bigoplus_{k:i<k \wedge j \neq k} Q_k C_k \oplus Q_{j\,u \in D_j}\{C_j(u) \oplus C_{ij}(v, u)\}$$

where ${Q_j}_{u\in D_j}\{C_j(u)\oplus C_{ij}(v,u)\}$ *is equal to:* $\min_{u\in D_j}\{C_j(u)\oplus C_{ij}(v,u)\}$ *if* $Q_j=\min$*, and* $\max_{u\in D_j}\{C_j(u)\oplus C_{ij}(v,u)\}$ *if* $Q_j=\max$.

Informally, this generic function is based on the NC lower bound $nc_{lb}(\mathcal{P}, x_i = v)$ and considers a binary constraint C_{ij} in addition. The only difference is that we need to pay extra attention to the unary cost of C_j. Quantifier Q_i will be used to determine prunings or backtrackings by referencing Table 4.1. For example, we are considering a $\min$ variable x_i and a $\max$ variable x_j. The function for AC can be easily derived from the general function: $ac_{lb}^{Q_i,Q_j}$. We obtain:

$$ac_{lb}^{\min,\max}(\mathcal{P}, x_i = v) \quad \equiv C_\varnothing \oplus C_i(v) \oplus \bigoplus_{k:i<k\wedge j\neq k} Q_k C_k \oplus \max_{u\in D_j}\{C_j(u)\oplus C_{ij}(v,u)\}$$

Theorem 4.11. $\forall v_1 \in D_1, ..., v_{i-1} \in D_{i-1}$, $\text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \geq ac_{lb}^{Q_i,Q_j}(\mathcal{P}, x_i = v)$.

Proof. (sketch) $C_\varnothing$ gives the global minimum costs, and $C_i(v)$ gives costs when value v is assigned to variable x_i. Including these two terms to the lower bound estimation is trivial as these costs must be incurred to all sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]$. Similar to NC, unary costs given by variables x_j where $j < i$ are ignored, as the minimum costs for unary constraints are 0 by enforcing weak-NC*. Consider the node corresponding to sub-problem $\mathcal{P}$ with $x_i = v$. If $Q_{i+1} = \max$, the cost of its child node is at least the maximum cost of C_{i+1}, since the $\max$ quantifier will choose a solution at least as great as $\max C_{i+1}$. Likewise, if $Q_{i+1} = \min$, the $\min$ quantifier will choose sub-problem with the minimum cost. The reasoning process continues recursively until the leaf node. Therefore, we can add unary costs given by $C_k, k > i$, including C_j, according to the quantifier Q_k. For the binary constraint C_{ij} on sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]$, we view it as a unary constraint C_j by assuming value v is assigned to variable x_i. Therefore, we merge the costs for the binary constraint C_{ij} with C_j using the expression $C_j(u)\oplus C_{ij}(v,u)$

for the approximation. The third term is modified to prevent double counting the unary constraint C_j. □

Upper Bound of A-Cost

Definition 4.12. *The function* $ac_{ub}^{Q_i,Q_j}$ *approximates the* A-cost *for the set* S *of sub-problems:*

$$\begin{aligned} ac_{ub}^{Q_i,Q_j}(\mathcal{P}, x_i = v) \equiv \;& C_\varnothing \oplus C_i(v) \oplus \bigoplus_{k:i<k \wedge j \neq k} Q_k C_k \oplus \bigoplus_{k<i} \max C_k \\ & \oplus \bigoplus_{C_{kl} \in B} \max C_{kl} \oplus Q_{j\,u \in D_j}\{C_j(u) \oplus C_{ij}(v,u)\} \end{aligned}$$

where $B = \{C_{kl} \in \mathcal{C} | k \neq l\} - \{C_{ij}\}$, $Q_{j\,u \in D_j}\{C_j(u) \oplus C_{ij}(v,u)\}$ *is equal to:* $\min_{u \in D_j}\{C_j(u) \oplus C_{ij}(v,u)\}$ *if* $Q_j = \min$, *and* $\max_{u \in D_j}\{C_j(u) \oplus C_{ij}(v,u)\}$ *if* $Q_j = \max$

Theorem 4.13. $\forall v_1 \in D_1, ..., v_{i-1} \in D_{i-1}$, $\text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \leq ac_{ub}^{Q_i,Q_j}(\mathcal{P}, x_i = v)$.

Proof. (sketch) Similar to the NC upper bound nc_{ub}, computing the upper bound is less straightforward, and we need to take *all* constraints into account. The function is based on the $nc_{ub}(\mathcal{P}, x_i = v)$ and considers a binary constraint C_{ij} in addition. The first five terms are adopted from the NC upper bound. The main difference is that we take account of the binary constraint C_{ij}. We apply the same technique in AC lower bound. For the binary constraint C_{ij} on sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]$, we can view it as a unary constraint C_j by assuming value v is assigned to variable x_i. We merge the costs for the binary constraint C_{ij} with C_j using the expression $C_j(u) \oplus C_{ij}(v,u)$ (sixth term) for the approximation. The third term and the fifth term are modified to prevent double counting the unary constraint C_j and the binary constraint C_{ij}. □

Projecting Binary Costs to Unary Constraint

To maximize propagation as in the case of NC, we will adopt part of AC* [23] in our Arc Consistency notion.

Definition 4.14 (Weak-AC*)**.**

$$\forall C_{ij}, \forall v_i \in D_i, \exists v_j \in D_j : C_{ij}(v_i, v_j) = 0$$
$$\forall C_{ij}, \forall v_j \in D_j, \exists v_i \in D_i : C_{ij}(v_i, v_j) = 0$$

Similar to NC, by enforcing weak-AC* using binary projections (discussed in the background section), we can make $nc_{lb}(\mathcal{P}, x_i = v)$/$ac_{lb}(\mathcal{P}, x_i = v)$ a tighter bound. However, weak-AC* does not improve nc_{ub}/ac_{ub} since the maximum cost of all constraints have already been taken into account.

Definition 4.15. *A QWCSP $\mathcal{P}$ is* Arc Consistent *(AC) when the following condition holds:*

$$\forall i, \exists v \in D_i : C_i(v) = 0 \quad \texttt{(Weak-NC*)}$$
$$\forall C_{ij}, \forall v_i \in D_i, \exists v_j \in D_j : C_{ij}(v_i, v_j) = 0 \ \wedge$$
$$\forall C_{ij}, \forall v_j \in D_j, \exists v_i \in D_i : C_{ij}(v_i, v_j) = 0 \texttt{(Weak-AC*)}$$
$$\forall C_{ij}, \forall v_i \in D_i : ac_{lb}^{Q_i,Q_j}(\mathcal{P}, x_i = v) < ub \ \wedge$$
$$\forall C_{ij}, \forall v_i \in D_i : ac_{ub}^{Q_i,Q_j}(\mathcal{P}, x_i = v) > lb$$

where ub and lb are the upper and lower bounds in alpha-beta pruning.

4.2.4 Enforcing Algorithm for AC

Similar to the NC enforcing algorithm, achieving AC in the QWCSP solver is easy. The solver would similarly maintain three conditions: 1) weak-AC* and weak-NC*, 2) $ac_{lb}^{Q_i,Q_j}(\mathcal{P}, x_i = v) < ub$, and 3) $ac_{ub}^{Q_i,Q_j}(\mathcal{P}, x_i = v) > lb$, in a step-wise manner. We use projection algorithms from WCSP solvers to maintain weak-AC* and weak-NC*. To maintain condition 2) and condition 3), we derive pruning/filtering algorithms based on the NC enforcing algorithm to prune values v of variables x_i which do not satisfy either condition 2) or condition 3). Writing such pruning/filtering algorithms can be seen as implementing functions `approx_lb` (line 11) and `approx_ub` (line 15) in Figure 4.2 by $ac_{lb}^{Q_i,Q_j}$ and $ac_{ub}^{Q_i,Q_j}$ respectively. Similar to the AC* enforcing algorithm for maintaining AC* in WCSPs, constraints may loss weak-AC* property after a value is pruned during enforcement of condition 2)/condition 3). To handle this situation, the propagation routine repeats (the while loop in lines 7 to 19) until no domains are further reduced.

We give a detailed combined AC and NC enforcing algorithm for the propagation routine in Figure 4.8. The enforcing algorithm `QWAC-1` is divided into projection phase (line 5 to 9) and pruning phase (line 10 to 33). Projection phase enforces weak-AC* and weak-NC* by performing projections, while the pruning phase enforces other remaining conditions by performing prunings/backtracks. To maximize pruning opportunities in the pruning phase, we enforce weak-AC* and weak-NC* by the projection phase first.

Projection Phase

The projection phase enforces weak-AC* by calling function `binaryProject`, which performs binary projections similar to function `FindSupportAC3` in the AC* enforcing algorithm for WCSPs. After enforcing weak-AC*, it will enforce weak-NC* by calling function `unaryProject`, which performs unary projections similar to the NC enforcing algorithm. Function `binaryProject` enhances `FindSupportAC3` by updating the data structure storing the maximum costs of a binary constraint (function `updateBinaryMaxCost` in line 54) if costs are being projected (line 41 and 49).

Pruning Phase

In the pruning phase, we enforce condition 2) and condition 3) by performing prunings. Function `QWAC-1-LbApprox` in Figure 4.9 and function

`QWAC-1-UbApprox` in Figure 4.10 show the pruning/filtering algorithms for enforcing condition 2) and condition 3) respectively. Again similar to the NC enforcing algorithm, the main goal of these two functions is to compute expression $ac_{lb}^{Q_i,Q_j}/ac_{ub}^{Q_i,Q_j}$ by gathering required costs from constraints. After the expressions are computed, bounds will then be checked and prunings/backtrackings will be performed accordingly.

Line 11 to line 21 shows the computation of arrays `MaxUnaryCost`, `SumOfUnaryCostsAfter`, and `SumofUnaryCostsBefore` for the expression $\max C_i$, $\bigoplus_{k:i<k} Q_k C_k$, and $\bigoplus_{k:k<i} \max C_k$ respectively. Details for these computations are stated in the NC enforcing algorithm.

The two functions both need to compute costs for the term $\bigoplus_{k:i<k \wedge j \neq k} Q_k C_k$, which is equivalent to this term: $\bigoplus_{k:i<k} Q_k C_k \ominus Q_j C_j$, assuming $\bigoplus_{k:i<k} Q_k C_k$ must have costs smaller than the global upper bound. Obtaining costs for the resulting term is easy, we can substract term $Q_j C_j$ from the ith element in the pre-comptued array `SumOfUnaryCostsAfter`. By weak-NC*, costs for the term $Q_j C_j$ must be 0 if $Q_j = \min$. Therefore, we only need to consider the substraction if $Q_j = \max$. Line 5 in Figure 4.9 (, and line 5 in Figure 4.10) shows the computation.

Function `QWAC-1-UbApprox` computes the term $\bigoplus_{C_{kl} \in B} \max C_{kl}$ in line 7, which sums up all the maximum costs for all binary constraints except the current checking binary constraint C_{ij}. We use a function `binaryMax()` to return the summation of all maximum possible costs given by all binary constraints, and also a function `maxCost(C)` to return the maximum costs of a constraint `C`.

After computing all the terms, the remaining for loops (`QWAC-1-LbApprox`: line 6 to 30, `QWAC-1-UbApprox`: line 8 to 34) compute the lower bound approximation (upper bound approximation resp.) according to the definition of $ac_{lb}^{Q_i,Q_j}$ ($ac_{ub}^{Q_i,Q_j}$ resp.). We then perform prunings/backtrackings accordingly.

Time complexity

Theorem 4.16. *`QWAC-1` runs in time $O(n^2d^2 + ned^3)$, where n is the number of variables, d is the maximum domain size, and e is the number of binary constraints in the corresponding QWCSP.*

Proof. (Sketch)

In the projection phase, we perform projections followed by pre-computing data structures for the pruning phase. We compute `unaryProject` for all unary constraints and the array `MaxUnaryCost` using time $O(nd)$, which have been shown in the NC enforcing algorithm. We also compute `binaryProject` for all binary constraints using time $O(ed^2)$. Recall in the NC enforcing algorithm, we pre-compute the two arrays: `SumOfUnaryCostsAfter` and `SumOfUnaryCostsBefore` for `QWNC-1-LbApprox` and `QWNC-1-UbApprox` in a recursive dynamic programming approach using time $O(n)$. These computations are re-used in the AC enforcing algorithm. We do not need to re-construct these two arrays in functions `QWAC-1-LbApprox` and `QWAC-1-UbApprox`.

The solver maintains a data structure which uses $O(e)$ space to store the maximum costs for each binary constraint, and $O(1)$ space to store the total sum of these maximum costs. We update the data structure during projections (in function `binaryProject`), and assignments. Therefore, the time being used to query `binaryMax()` and `maxCost()` for a binary constraint are both $O(1)$.

It is not hard to observe the time complexity of function `QWAC-1-LbApprox` and `QWAC-1-UbApprox` are both time $O(ed^2)$, as these algorithms calculate costs by scanning tuples in binary constraints. We can conclude the time complexity for running the propagation routine once is $O(nd + ed^2)$. The propagation routine may restart when a value of a variable is pruned, and there are $O(nd)$ values. Overall, the time complexity of `QWAC-1` is $O(n^2d^2 + end^3)$.

□

```
function QWAC-1(P)
  change = true
  while change:
    change = false
    // Projection Phase
    for Cij in C:
      binaryProject(Cij)
    for i in [1..n]:
      unaryProject(Ci)
    // Pruning Phase
    for i in [1..n]:
      MaxUnaryCost[i] = maxCost(Ci)
    for i in [1..n]:
      SumOfUnaryCostsAfter[i] = 0
      for k in [i+1..n]:
        if Qk == max:
          SumOfUnaryCostsAfter[i] += MaxUnaryCost[k]
    for i in [1..n]:
      SumOfUnaryCostsBefore[i] = 0
      for k in [1..i-1]:
        SumOfUnaryCostsBefore[i] += MaxUnaryCost[k]
    if QWAC-1-LbApprox(P,M):
      change = true
      if domainWipeOut(): return
    if QWNC-1-LbApprox(P,M):
      change = true
      if domainWipeOut(): return
    if QWAC-1-UbApprox(P,M):
      change = true
      if domainWipeOut(): return
    if QWNC-1-UbApprox(P,M):
      change = true
      if domainWipeOut(): return
  return
function binaryProject(Cij)
  change = false
  for u in Di:
    minCost = k
    for v in Dj:
      if Cij(u,v) < minCost: minCost = Cij(u,v)
    if minCost > 0: change = true
    Ci(u) = Ci(u) + minCost
    for v in Dj:
      Cij(u,v) = Cij(u,v) - minCost
  for v in Dj:
    minCost = k
    for u in Di:
      if Cij(u,v) < minCost: minCost = Cij(u,v)
    if minCost > 0: change = true
    Cj(v) = Cj(v) + minCost
    for u in Di:
      Cij(u,v) = Cij(u,v) - minCost
  if change:
    updateBinaryMaxCost(Cij)
```

Figure 4.8: Enforcing algorithm for AC

```
function QWAC-1-LbApprox(P,M)
  change = false
  for Cij in C:
    SumOfUnaryCosts = SumOfUnaryCostsAfter[i]
    if Qj = max: SumOfUnaryCosts -= MaxUnaryCost[j]
    for u in Di:
      if Qj = max:
        maxCost = 0
        for v in Dj:
          if Cj(v) + Cij(u,v) > maxCost: maxCost = Cj(v) + Cij(u,v)
        if C_NULL + Ci(u) + SumOfUnaryCosts + maxCost >= ub:
          if Qi = min:
            prune(xi,v)
            change = true
          if Qi = max:
            pruneAll(xi)
            change = true
            return change
      if Qj = min:
        minCost = k
        for v in Dj:
          if Cj(v) + Cij(u,v) < minCost: minCost = Cj(v) + Cij(u,v)
        if C_NULL + Ci(u) + SumOfUnaryCosts + minCost >= ub:
          if Qi = min:
            prune(xi,v)
            change = true
          if Qi = max:
            pruneAll(xi)
            change = true
            return change
  return change
```

Figure 4.9: AC lower bound approximation

```
function QWAC-1-UbApprox(P,M)
  change = false
  for Cij in C:
    SumOfUnaryCosts = SumOfUnaryCostsAfter[i]
    if Qj = max: SumOfUnaryCosts -= MaxUnaryCost[j]
    SumOfUnaryCosts += SumOfUnaryCostsBefore[i]
    otherBinaryMax = binaryMax() - maxCost(Cij)
    for u in Di:
      if Qj = max:
        maxCost = 0
        for v in Dj:
          if Cj(v) + Cij(u,v) > maxCost: maxCost = Cj(v) + Cij(u,v)
        if C_NULL + Ci(u) + SumOfUnaryCosts +
           otherBinaryMax + maxCost <= lb:
          if Qi = max:
            prune(xi,v)
            change = true
          if Qi = min:
            pruneAll(xi)
            change = true
            return change
      if Qj = min:
        minCost = k
        for v in Dj:
          if Cj(v) + Cij(u,v) < minCost: minCost = Cj(v) + Cij(u,v)
        if C_NULL + Ci(u) + SumOfUnaryCosts +
           otherBinaryMax + minCost <= lb:
          if Qi = max:
            prune(xi,v)
            change = true
          if Qi = min:
            pruneAll(xi)
            change = true
            return change
  return change
```

Figure 4.10: AC upper bound approximation

Chapter 5

Performance Evaluation

This chapter evaluates the performance of our solving techniques. To show the effectiveness of our approach, we compare our framework and techniques against one of our related works QCOPs. QWCSPs are special cases of QCOPs/QCOPs+ [5]. Given a QWCSP, we can show how to construct a QCOP instance by the "Soft As Hard" approach [35], which was used to construct classical COP instances from WCSP instances. We then conjecture, though QWCSPs are special instances of QCOP, our consistency notions provide more pruning opportunities than those for QCOP/QCOP+ when tackling the same QWCSP, and illustrate this using an example. In the remaining part of the chapter, we provide empirical evidence to demonstrate that our proposed solving techniques are more efficient than those for QCOPs for tackling QWCSPs.

5.1 Definitions of QCOP/QCOP+

We first give the definitions of QCOP/QCOP+ [5]. Given a set of variables V as in classical CSPs. A *restricted quantified set of variables* (rqset) is a tuple (q, W, C) where $q \in \{\exists, \forall\}$, W is a subset of variables where $W \subseteq V$, and C is a CSP. A *prefix* P of rqsets is a sequence of rqsets $((q_1, W_1, C_1), \ldots, (q_n, W_n, C_n))$ such that $W_i \cap W_j = \varnothing, \forall i \neq j$. We define *range(P)* for a prefix P with n rqsets to be $[1..n]$, *before*$_i$*(P)* to be $\bigcup_{j \leq i} W_j$, and $nu_i(P) = \min_{j>i}\{j | q_j = \forall\}$ to be the index of the next universal block of variables located after an index i. If no such index exists, we denote $nu_i(P)$ by $n+1$.

Example 20. *Given a prefix of rqsets P as follows:*

$$((\exists, \{x_1, x_2\}, C_1),(\forall, \{x_3\}, C_2), (\exists, \{x_4\}, C_3)), \text{ where}$$
$$C_1 = (\{x_1, x_2\}, \{D_1 = \{1,2\}, D_2 = \{2,3\}\}, \{x_1 \neq x_2\}),$$
$$C_2 = (\{x_1, x_3\}, \{D_1 = \{1,2\}, D_3 = \{1,3\}\}, \{x_1 \neq x_3\}), \text{ and}$$
$$C_3 = (\{x_3, x_4\}, \{D_3 = \{1,3\}, D_4 = \{2,3\}\}, \{x_4 \neq 3, x_4 < x_3\})$$

In a QCOP/QCOP+, if variables with the same indices appear in different CSPs C_i of a prefix of rqsets, we assume these variables are common, i.e. they are referring to the same variable with the same domain. In P, we can see variable x_1 appears in both CSP C_1 and C_2, and variable x_3 appears in both CSP C_2 and C_3. To simplify writing, we write P as follows:

$$\begin{array}{llll}
\exists\ x_1 \in \{1,2\} & x_2 \in \{2,3\} & [x_1 \neq x_2] & \\
 & \forall\ x_3 \in \{1,3\} & [x_1 \neq x_3] & \\
 & & \exists x_4 \in \{2,3\} & [x_4 \neq 3, x_4 < x_3]
\end{array}$$

We can easily see that the range of P is $[1..3]$ as there are three rqsets, before$_2(P) = W_1 \cup W_2$ is the set of variables $\{x_1, x_2, x_3\}$, $nu_1(P)$ is equal to 2 as $q_2 = \forall$, and $nu_2(P)$ is equal to 4.

Given two winning strategies of a QCSP, we cannot tell which one is better. The reason behind is that we do not quantify how good a winning strategy is. To quantify a winning strategy, one way is to give different costs to different scenarios. However, for a winning strategy, there are many scenarios, and we have to aggregate these costs for each of these scenarios. To classify winning strategies, some problems require taking the costs for the worst case scenario, while some may need to sum up all the costs for all scenarios. QCOP/QCOP+ allows different aggregation functions associate with aggregate names to tackle this issue. We can choose max functions to aggregate costs for the worst case scenario, and define aggregate names to label these aggregated costs. An aggregate function f is defined by an associative function on a multiset of values, and a neutral element 0_f which indicates the value of $f(\{\!\{\}\!\})$. Let $\mathcal{A}$ be a set of aggregate names and $\mathcal{F}$ be a set of aggregate functions. An *aggregate* is an atom of the form $a : f(X)$ where $a \in \mathcal{A}$, $f \in \mathcal{F}$, and $X \in V \cup \mathcal{A}$. Intuitively, we can see that aggregate functions aggregate values of variables in V as well as aggregate names. However, we are not allowed to use variables in V to store the aggregated value of an aggregate function.

An *optimization condition* is an atom of the form $\min(X)$, $\max(X)$ where $X \in V \cup \mathcal{A}$ or the atom any. An atom $\min(X)$ ($\max(X)$ resp.) means the user is interested in strategies that minimize (maximize resp.) this value, while any indicates the user does not care about the returned strategy.

An $\exists$-*orqset* is a tuple $(\exists, W, C, o)$ where $(\exists, W, C)$ is a rqset and o is an optimization condition. A $\forall$-*orqset* is a tuple $(\forall, W, C, A)$ where $(\forall, W, C)$ is a rqset and A is a set of aggregates (of the form $a : f(X)$). We denote *names* of the set of aggregates A in a $\forall$-orqset by $names(A)$. An *orqset* is either a $\exists$-orqset or a $\forall$-orqset.

Given a QCOP+. Suppose the value of the variable x_1 in an $\exists$-orqset is subject to change for different value assignments of current set of variable W in an $\forall$-orqset, which is located before the $\exists$-orqset. We can define an aggregate function `sum` which sums the values of x_1 for different value assignments of the current set of variables in W. We write $a : \texttt{sum}(x_1)$ in the set of aggregates A in the $\forall$-orqset to mean a will take the sum of all values taken by variable x_1, for all combinations of assignments for the set of variables W.

A *QCOP+* is a pair (P, G) where G is a CSP and $P = (orq_1, \ldots, orq_n)$ is a prefix of orqsets such that $\forall i \in range(P)$, with $k = nu_i(P)$:

(1) if $orq_i = (\exists, W, C, o)$ with $o = \min(X)$ or $o = \max(X)$, then we must have $X \in$ ***before***$_{k-1}(P) \cup (k < n + 1 ? names(A_k) : \varnothing)$, and

(2) if $orq_i = (\forall, W, C, A)$, then for all a : f(X) in A, we must have $X \in$ *before*$_{k-1}(P) \cup$ $(k < n + 1?names(A_k) : \varnothing)$.

A QCOP is a QCOP+ in which no orqsets have restrictions, *i.e.* no constraints in the CSP of all orqsets.

Example 21. *We begin by giving the example QCOP+ (P, G) in [5], with slight modifications as follows. Given an array A of integers ranging from 0 to 100. P is a prefix of orqsets:*

$$((\exists, \{x_1\}, C_1, o_1), (\forall, \{x_2\}, C_2, A_2), (\exists, \{x_3\}, C_3, o_3)) \text{ , where}$$

$$\begin{aligned}
C_1 =&(\{x_1\}, \{D_1 = \{0, 1\}\}, \emptyset),\\
C_2 =&(\{x_1, x_2\}, \{D_1 = \{0, 1\}, D_2 = [0..9]\}, \{x_2 \mod 2 = x_1\}),\\
C_3 =&(\{x_3\}, \{D_3 = [0..100]\}, \emptyset),\\
o_1 =&\min(a_1),\\
A_2 =&\{a_1 : \texttt{sum}(x_3)\}, \text{ and}\\
o_3 =&any
\end{aligned}$$

G is a classical CSP as follows:

$$G = (\{x_2, x_3\}, \{D_2 = [0..9], D_3 = [0..100]\}, \{x_3 = A[x_2]\})$$

To simplify writing, we write the QCOP+ as follows:

$$\begin{array}{lll}
\exists \ x_1 \in \{0, 1\} & & \\
& \forall \ x_2 \in [0..9][x_2 \mod 2 = x_1] & \\
& & \exists \ x_3 \in [0..100] \\
& & x_3 = A[x_2] \\
& a_1 = \texttt{sum}(x_3) & \\
\min(a_1) & &
\end{array}$$

We ignore all optimization conditions which are any.

To define the semantic of a QCOP+, we need to define several notations. Given two subset of variables W and U. We define their sets of possible combinations of values be $D^W = \{\times D_i | x_i \in W\}$, $D^U = \{\times D_j | x_j \in U\}$. Naturally, the set of possible combinations of values for the set of variable $W \cup U$ is the set $D^{W \cup U} \{\times D_i | x_i \in W \cup U\}$. Let $A \subseteq W$ and $B \subseteq U$. We define *join* ($A \bowtie B$) on A and B is the set $\{t \in D^{W \cup U} | t[W] \in A \wedge t[U] \in B\}$. We can view $A \bowtie B$ as a set constructed from A and B indicating the subset of possible combinations of values for the set of variables W and V. We denote by $|$ the sequence constructor and by $()$ the empty sequence. For example, we use $(orq_1 | (orq_2, orq_3))$ to construct a sequence of orqsets (orq_1, orq_2, orq_3). Given a QCOP+ $Q = (P, G)$. Let $sol(C)$ be the set of solutions for the classical CSP C, and $var(C)$ be the set of variables in the classical CSP C. We define function val computing the value of an aggregate $a : f(X)$ according to a set s, which contains tuples of values, as follows: $val(a : f(X), s) = f(\{\!\{t[X] | t \in s\}\!\})$. We first define strategies [5] for the prefix P of a QCOP/QCOP+ $Q = (P, G)$ as follows:

$$\begin{aligned} STRAT(()) =& \emptyset \\ STRAT(((\exists, W, C) | P')) =& \{t \bowtie s' | t \in D^W \wedge s' \in STRAT(P')\} \\ STRAT(((\forall, W, C) | P')) =& \{\bigcup \alpha(D^W) | \alpha \in \prod_{t \in D^W} (\{t \bowtie s' | s' \in STRAT(P')\})\} \end{aligned}$$

If the prefix is empty, the set of strategies is an empty set. Given that the first orqset is a $\exists$-orqset. The possible set of strategies is the possible combinations of values for the set of variables W, joined by the sub-strategies s' for the remaining prefixes. Given that the first orqset is a $\forall$-orqset. The term $\{t \bowtie s' | s' \in STRAT(P')\}$ takes account of all sub-strategies for each combiantion t of values for the set of variables W. It computes the set of strategies beginning by a tuple t. The operator $\prod_{t \in D^W}$ takes the cartesian product of all these sets, returning a set of tuples. Each tuple α specifies a strategy for all combinations t for the set of variables, and therefore, an α specifies $|D^W|$ strategies. We can also view α as a function called $\alpha(D^W)$ mapping each tuple of D^W to a strategy, which is also a set of tuples. Suppose for each function $\alpha(D^W)$, we take the union of the image set. The union is a new strategy which contains a sub-strategy for each $t \in D^W$. The goal is to construct the union of the image set for each function $\alpha(D^W)$.

The *optimal strategies* $WIN(Q)$ for the QCOP+ Q is defined recursively as follows:

- Base case where there are no orqsets:

  $$WIN(((), G)) = sol(G)$$

  Optimal strategies of the QCOP are equal to the set of solutions of the body CSP G.

- First orqset is an $\exists$-orqset without any optimization conditions:

  $$WIN((((\exists, W, C, any)|P'), G)) =$$
  $$\{t \bowtie s | t \in D^W \wedge t[var(C)] \in sol(C) \wedge s \in WIN((P', G))\}$$

  The optimal strategies are equal to joining the optimal strategies s without the first orsqet, and combinations of values t which satisfy the CSP C.

- First orqset is an $\exists$-orqset with minimization condition:

  $$WIN((((\exists, W, C, \min(X))|P'), G)) =$$
  $$\{s \in WIN((((\exists, W, C, any)|P'), G)) | s[X] = \min_{s' \in WIN((((\exists, W, C, any)|P'), G))} (s'[X])\}$$

  The optimal strategies must be the optimal strategies with any as the optimization condition. We further minimize these strategies according to the minimization condition $\min(X)$.

- First orqset is an $\exists$-orqset with maximization condition:

  $$WIN((((\exists, W, C, \max(X))|P'), G)) =$$
  $$\{s \in WIN((((\exists, W, C, any)|P'), G)) | s[X] = \max_{s' \in WIN((((\exists, W, C, any)|P'), G))} (s'[X])\}$$

The optimal strategies must be the optimal strategies with any as the optimization condition. We further maximize these strategies according to the maximization condition $\max(X)$.

- First orqset is a $\forall$:

$$\begin{aligned}
&WIN((((\forall, W, C, A)|P'), G)) = \\
&\quad \{(val(a : f(X), s))_{a\in names(A)} \bowtie s | s \in WIN((((\forall, W, C)|P'), G))\} \\
&WIN((((\forall, W, C)|P'), G)) = \\
&\qquad \{\bigcup \alpha(D^W) | \alpha \in \prod_{t\in D^W} (\{t \bowtie s | t[var(C)] \in sol(C)? \\
&\qquad\quad s \in WIN(P', G) : s \in STRAT(P', G)\})\}
\end{aligned}$$

Intuitively, we can view the optimal strategies are a set of optimal sub-strategies, where each sub-strategies represent strategies w.r.t. a combination of values for the variables in set W. We then compute all the aggregates and stores the result in the optimal strategies.

However, we have to deal with cases where the combination t of values for variables in W do not satisfy the constraint C. It follows that any sub-strategies s for all these violating combinations can be freely glued, and we require these strategies s in the form of $STRAT(P', G)$.

Given a QCOP+, the goal is to find its optimal strategy. We omit the Branch and Bound tree search for solving QCOP+. The framework is general enough to model QCSPs, and even bi-level programs [5]. It is natural for this framework to model our proposed framework, QWCSPs.

In Example 21, we can see that variable x_1 can choose value 0 or 1, and its objective is to minimize the aggregate name a_1. If x_1 is assigned to 0 (1 resp.), x_2 must take even (odd resp.) numbers in the interval $[0..9]$. The aggregate name a_1 is summing even (odd resp.) indices of the array A, as x_3 must be assigned to

$A[x_2]$. Therefore, x_1 will choose 0 if the summation of odd indicies in the array A is smaller than or equal to the summation of even indicies.

5.2 Transforming QWCSPs into QCOPs

We can transform any QWCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}, k)$ into a QCOP $\mathcal{P}' = (P', G')$ based on the modified "Soft As Hard" approach as follows. For each variable x_i in $\mathcal{P}$, there is an orqset $orq_i = (\exists, \{x'_i\}, C'_i, o'_i)$ in P', where C'_i has no constraints. For every soft constraint C in $\mathcal{C}$, there is a corresponding cost variable x'_c in the CSP G' with domain being equal to all possible costs given by C. We construct an auxiliary cost variable s in CSP G' which is equal to the sum of all cost variables x'_c. A constraint $s < k$ is added to restrict the total cost to be less than the global upper bound. If x_i is a minimization variable, then we add $o'_i = \min(s)$; otherwise, $o'_i = \max(s)$. Suppose C on a set S of variables giving cost m when a tuple of assignment l is assigned. There is a *reified constraint* in the CSP G' restricting x'_c to take value m if variables in S are assigned with tuple l.

Example 22. *Given a QWCSP $\mathcal{P}$ with an ordered sequence of variables (x_1, x_2), domains $D_1 = D_2 = \{a, b\}$, a set of constraints $\{C_1, C_2\}$, a quantifier sequence $\{Q_1 = \min, Q_2 = \max)$, and a global upper bound $k = 7$. $C_1(a) = 0, C_1(b) = 5, C_2(a) = 1, C_2(b) = 3$. The QCOP $\mathcal{P}'$ can be expressed using the "Soft As Hard" approach as follows:*

$$\begin{array}{lll}
\exists x'_1 \in D_1 & & \\
 & \exists x'_2 \in D_2 & \\
 & & x'_{C_1} \in \{0,5\}, x'_{C_2} \in \{1,3\}, s \in \{1,3,6\} \\
 & & s = x'_{C_1} \oplus x'_{C_2} \wedge s < 7 \\
 & & [x'_1 = a] \implies [x'_{C_1} = 0] \wedge [x'_1 = b] \implies [x'_{C_1} = 5] \\
 & & [x'_2 = a] \implies [x'_{C_2} = 1] \wedge [x'_2 = b] \implies [x'_{C_2} = 3] \\
 & \max s & \\
\min s & &
\end{array}$$

Theorem 5.1. *A QWCSP $\mathcal{P}$ can be transformed into a QCOP $\mathcal{P}'$. The* $\mathrm{A\text{-}cost}$ *of P can be found by solving the optimal strategy [5] of $\mathcal{P}'$.*

The proof follows directly from the “Soft As Hard”construction. In particular, $\mathrm{A\text{-}cost}$ of $\mathcal{P}$ is equal to the value assigned to s in the optimal strategy [5] of $\mathcal{P}'$.

The central theme of this paper is to show QWCSP, a more restricted but useful subclass of QCOP+, can be solved more efficiently. Based on WCSPs, QWCSP’s consistency techniques redistribute constraint costs by projections and extensions. In turn, we conjecture that this allows extra prunings over the classical consistencies used in QCOP+. The situation is similar to how Lee and Leung [24] show WCSP (soft approach) consistencies to be stronger than classical optimization used in “Soft As Hard”. We illustrate this idea using an example. In Example 22, the QWCSP $\mathcal{P}$ is not node consistent as value b of x_1 is not NC. QCOP+ performs cascade propagation [5] by utilizing standard propagators in CSPs. In the corresponding $\mathcal{P}'$, the constraints in the body part is NC and GAC [1] in classical CSPs.

5.3 Empirical Evaluation

In this section, we compare the QCOP+ solver QeCode against our solver in three progressive modes: Alpha-beta pruning, Node Consistency (NC), and Arc Consistency (AC). Values are labeled in three ways: static lexicographic order and two dynamic value ordering heuristics based on value costs.

We generate 20 instances for each benchmark’s particular parameter setting. Results for each benchmark are tabulated with number of solved instances, average time used, and average number of tree nodes encountered. We take average for solved instances *only*. Winning entries for average time used and average number of tree nodes encountered are highlighted in bold. A symbol ‘-’ represents all instances fail to run within the time limit of 900 seconds. The experiment is conducted on a Pentium 4 3.2GHz with 3GB memory.

We compare our solver against QeCode, which uses minimax. All QWCSP instances are transformed to QCOP using the “Soft As Hard” approach outlined. We note alpha-beta prunings can be employed for QCOP+, but we believe there will be less prunings by enforcing classical consistencies.

5.3.1 Random Generated Problems

We generate random binary QWCSPs with parameters (n, s, d), where n is the number of variables, s is the domain size for each variable, and d is the probability for a binary constraint to occur between two variables. We purposefully do not generate unary constraints to make the problem harder to solve. The costs for each binary constraint are generated uniformly in [0..30]. Quantifiers are generated randomly with half probability for min (max resp.), and number of quantifier levels vary from instances to instances.

Table 5.1 shows the results. For all instances, even just alpha-beta pruning is two orders of magnitude faster than QeCode, which cannot handle even moderately sized instances. NC and AC both run faster than alpha-beta pruning, with the search space dramatically decreased and runtime significantly faster. AC, utilizing information from both unary and binary constraints, performs best among all solver modes.

5.3.2 Graph Coloring Game

We have generated instances (v, c, d) for a graph coloring game similar to the one in the introduction, where v is an even number of nodes in the graph, c is the range of numbers allowed to place, and d is the probability of an edge between two vertices. Player 1 (Player 2 resp.) is assigned to play the odd (even resp.) numbered turns, players play in turn, and in each turn a node is numbered. The node corresponding to each turn is generated randomly.

There are v variables in the QWCSPs, each of them has a domain of $[1..c]$ representing a node in the graph. If there is an edge between two nodes, there is a binary constraint constraining the two nodes. Quantifier Q_i is min (or max resp.) if turn i is played by player 1 (player 2 resp.). The violation measure of a binary constraint C_{ij} is $(c-1) - |x_i - x_j|$ (, or equivalently $|x_i - x_j|$ by swapping the $\min$ and $\max$ quantifiers).

Table 5.2 shows the results. Similar to Random Problems, alpha-beta pruning runs faster than QeCode in all instances two orders of magnitude faster. NC and AC both run faster than alpha-beta pruning. Comparing AC and alpha-beta pruning, we can gain up to six times speedup for AC, which prunes up to two-third of the search space of NC. Again, AC betters in almost all instances in runtime.

5.3.3 Min-Max Resource Allocation Problem

Suppose N units of resources are allocated to t activities. We let x_i be the amount of resources allocated to activity i, and a function $c_i(x_i)$ returns the cost incurred from activity i by allocating x_i units of resources the activity. The resource allocation problem [28] is to find an optimal resource allocation so as to minimize the total cost. Suppose now there are s functions $c_i^1, c_i^2, \ldots, c_i^s$. The min-max resource allocation problem [48] is to find a resource allocation to minimize the maximum cost functions.

We have t variables $x_i, 1 \leq i \leq t$ in the QWCSP associated with domains $[1..N]$. Each variable x_i represents the amount of resources allocated to activity i. There is a hard linear constraint restricting the summation of these variables must be smaller than or equal to N. There is another variable x_p with a domain $[1..s]$, which represents which function we are choosing. If x_p is assigned to 1, we are choosing function c_i^1. For each variable x_i, we have a binary constraint $C_{ip}(a, b)$ constraining on variable x_i and variable x_p. The constraint gives costs for the function c_i^b when a unit of resources is allocated to activity i. There are no particular requirements on the variable ordering between the t variables $x_i, 1 \leq i \leq t$, except we must ordere x_p as the last variable.

Table 5.3 shows the results. Alpha-beta pruning runs an order of magnitude faster than QeCode. Again, NC and AC both run faster than alpha-beta pruning. AC can prune up to 90% of the search space of NC and runs the fastest in all instances.

5.3.4 Value Ordering Heuristics

Value ordering heuristics are useful for solving WCSPs [17, 25], QCSPs [42], and QCOPs+ [45]. In this section, we propose two value ordering heuristics for our solver, which are inspired by adversarial search (minimax heuristics).

For a sub-problem QWCSP $\mathcal{P}$ at a tree node, the goal of the two heuristics is to choose values v for the next unassigned $\min$ ($\max$ resp.) variables x_i with lower (higher resp.) $\text{A-cost}(\mathcal{P}[x_i = v])$. Instead of computing the exact costs of sub-problems, which are expensive, these two heuristics estimate costs from constraints associated with x_i. The first heuristic checks costs from the unary constraint C_i, while the second heuristic checks costs from all unary and binary constraints covering x_i. Our approach is similar to solution-focused approach [42] for QCSPs. For $\min$ ($\max$ resp.) variables, heuristics *ADVUnary* will choose value $v \in D_i$ of the next unassigned variable x_i if $C_i(v)$ is the smallest (largest resp.); while heuristic *ADVBinary* will choose $v \in D_i$ of the next unassigned variable x_i if $C_i(v) \oplus \bigoplus_{k>i} Q_{k u \in D_k} C_{i,k}(v, u)$ is the smallest (largest resp.).

We give preliminary results on the above three benchmarks, by comparing the two proposed ordering heuristics with the lexicographic ordering (Static). All experiments run with AC. We mark ADVBinary the best for instance $(18, 5, 0.4)$ in Table 5.1 as ADVBinary solves the most number of instances. Table 5.2 shows ADVBinary runs fastest on the Graph Coloring Game, and Table 5.1 shows ADVUnary and ADVBinary runs faster than Static for Random Generated Problem. For Min-Max Resource Allocation Problem in Table 5.3, Static runs faster than the other two heuristics. The results vary. We can only conclude that value ordering does matter but how well certain heuristics work depends on particular problem characteristics.

	QeCode			Alpha-beta			Node Consistency			Arc Consistency								
				Static			Static			Static			ADVUnary			ADVBinary		
(n, s, d)	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes
(9, 5, 0.4)	20	401.62	4,394,531	20	0.84	110,738	20	0.19	11,115	20	0.17	4,221	20	**0.05**	545	20	**0.05**	**539**
(9, 5, 0.6)	20	550.32	4,394,531	20	1.95	228,017	20	0.23	11,817	20	0.28	4,024	20	0.11	855	20	**0.10**	**748**
(12, 5, 0.4)	0	-	-	20	91.23	5,967,461	20	5.04	158,179	20	4.25	53,866	20	0.91	5,775	20	**0.76**	**4,623**
(12, 5, 0.6)	0	-	-	20	66.54	4,782,541	20	3.87	118,40	20	4.27	41,710	20	1.30	6,975	20	**1.26**	**6,385**
(16, 5, 0.4)	0	-	-	2	706.51	26,269,025	20	299.15	5,780,075	20	193.63	1,657,203	20	20.62	86,855	20	**17.88**	**71,379**
(16, 5, 0.6)	0	-	-	1	830.29	32,859,735	17	438.42	8,085,200	19	428.35	2,909,388	20	32.96	112,895	20	**30.94**	**100,711**
(18, 5, 0.4)	0	-	-	0	-	-	1	815.69	11,210,135	3	595.33	3,514,426	17	150.68	505,905	18	**156.91**	**502,324**
(18, 5, 0.6)	0	-	-	0	-	-	1	718.15	9,171,827	2	470.27	2,613,430	18	104.58	316,380	18	**99.87**	**287,408**

Table 5.1: Random Generated Problem

	QeCode			Alpha-beta			Node Consistency			Arc Consistency								
				Static			Static			Static			ADVUnary			ADVBinary		
(v, c, d)	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes
(10, 4, 0.4)	20	141.16	2,446,677	20	0.41	43,509	20	0.13	6,158	20	0.11	2,580	20	**0.03**	413	20	**0.03**	**299**
(10, 4, 0.6)	20	179.58	2,446,677	20	0.49	49,029	20	0.17	8,124	20	0.17	3,617	20	0.06	553	20	**0.04**	**309**
(12, 4, 0.4)	0	-	-	20	3.46	266,589	20	0.95	33,739	20	0.70	13,124	20	0.17	1,810	20	**0.10**	**990**
(12, 4, 0.6)	0	-	-	20	4.22	302,255	20	1.34	47,010	20	1.19	18,088	20	0.25	1,924	20	**0.13**	**929**
(16, 4, 0.4)	0	-	-	20	214.93	10,050,800	20	44.73	1,002,145	20	30.20	363,523	20	2.58	17,399	20	**1.21**	**7,590**
(16, 4, 0.6)	0	-	-	20	210.21	9,213,029	20	43.85	949,861	20	37.93	352,691	20	3.58	18,226	20	**1.27**	**5,569**
(18, 4, 0.4)	0	-	-	0	-	-	20	278.00	4,095,993	20	158.71	1,315,212	20	11.60	58,268	20	**4.22**	**19,239**
(18, 4, 0.6)	0	-	-	0	-	-	20	362.04	5,295,433	20	238.51	1,711,880	20	14.97	60,666	20	**4.62**	**15,657**

Table 5.2: Graph Coloring Game

	QeCode			Alpha-beta			Node Consistency			Arc Consistency								
				Static			Static			Static			ADVUnary			ADVBinary		
(t, N, s)	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes	#solve	Time	#nodes
(10, 8, 7)	20	49.03	656,221	20	2.17	146,778	20	1.13	39,605	20	**0.48**	**4,317**	20	0.55	4,965	20	0.56	4,965
(12, 8, 7)	20	166.42	1,889,371	20	9.26	444,340	20	4.95	132,257	20	**1.91**	**15,096**	20	2.40	19,355	20	2.44	19,355
(12, 9, 7)	20	399.39	4,408,771	20	20.63	981,967	20	8.61	226,456	20	**3.05**	**21,563**	20	3.80	28,137	20	3.86	28,137
(13, 9, 7)	20	716.22	7,461,106	20	41.82	1,732,234	20	17.17	398,072	20	**5.87**	**39,831**	20	6.85	47,118	20	6.95	47,118
(13, 9, 8)	20	852.29	8,455,920	20	42.07	1,700,088	20	17.25	399,221	20	**6.34**	**39,526**	20	7.56	48,103	20	7.67	48,103

Table 5.3: Min-Max Resource Allocation

Chapter 6

Concluding Remarks

In this chapter, we summarize our works and contributions. Limitations, related works, and possible future enhancements for our framework will also be discussed.

6.1 Contributions

We define a QWCSP framework for modeling optimization problems with adversaries. The framework is based on WCSPs by adding $\max$ quantifiers. We give formal definitions and semantics to our framework, and show the relationships between WCSPs and QCSPs.

We propose and implement a complete solver incorporating alpha-beta prunings into branch-and-bound. We formally give sufficient pruning conditions which allow us to prune the search space of QWCSPs, via approximating the lower/upper bound for a set of sub-problems. Correctness for these pruning conditions follows naturally from alpha-beta prunings. We then devise Node Consistency and Arc Consistency, based on these pruning conditions and exploiting the quantifier semantics, to allow further pruning of search space. Techniques from WCSPs are reused in order to allow Node Consistency and Arc Consistency performing bounds approximation. Enforcing algorithms with time complexities for both Node Consistency and Arc Consistency are discussed.

We show that QCOP is indeed more general than our framwork, and conjecture that though our framwork is more specific, consistency methods developed for our framework is stronger than those for QCOP. Our work allows us to model and solve interesting problems, such as graph coloring game and min-max resource allocation problem efficiently.

We evaluate our proposal using three benchmarks and compare with QeCode. Two value ordering heuristics are proposed. Experimental results show consistency enforcement is worthwhile in general and our solver is orders of magnitude faster than QeCode in tackling QWCSPs. We also gain some understanding about value ordering heuristics.

6.2 Limitations and Related Works

QWCSPs attempt to explore soft constraint techniques combining with alpha-beta prunings to tackle several classes of optimization problems with adversaries. In this thesis, we assume adversaries always choose the worst case scenario for the users/players. Therefore, the adversaries are worst-case adversaries. In general, an adversary may be an average-case adversary, or the adversary may be interested in optimizing his/her costs only.

There are other works exploring in different directions, which can also tackle these problems. The Plausibility-Feasibility-Utility (PFU) framework [36] is a general framework, which formalizes sequential decision problems with feasibility (restriction of descisions), utility (cost), and plausibility (uncertainty). We can view QWCSPs as a sub-class of the PFU framework, by expressing feasibility and utility as soft constraint and plausibility as $\max$ quantifier.

Another related work is Stochastic CSPs (SCSPs) [47], which can represent adversaries by known probability distributions. Probabilities for every action decided by adversaries are known beforehand. We then seek actions to minimize/maximize the expected cost for all possible scenarios. Our work is similar in the sense that we are minimizing the cost for the worst case scenario. At current stage, QWCSPs do not handle probability distributions for any of the adversaries.

QCSP+ [4]/QCOP+ [5] are also our related works. QCSP+ enhances QCSP by allowing restricted quantified set of variables (rqsets) associated with sets of variables. Benedetti et. al. then give the optimization framework QCOP/QCOP+ at later times. QCSP+/QCOP+ allows us to model practical applications, which contain restrictions restricting moves(, or decisions) of a player conditioned on previous moves(, or decisions). Chapter 5 shows the relationship between QCOP/QCOP+ and QWCSPs.

Bilevel programming [13] is a mathematical programming framework allowing us to model "mathematical programs with optimization problems in the constraints", in which it is useful in solving problems in the form of Stackelberg games/Stackelberg leadership models [46]. Stackelberg leadership model is a strategic game (in economics) in which the leader firm moves first and the follower firms move afterwards. We can translate the game, in game-theoretical terminologies, as a two-round game, where the first round is played by a leader (representing the leader firm), followed by the second round played by a follower (representing the follower firms), both aim to maximize their profits. Most often, we want to find the subgame perfect Nash equilibrium [32] of a Stackelberg game, which states the best responses for the leader and the follower. We can also use QWCSPs to find the subgame perfect Nash equilibrium for two-player ($\min$ and $\max$ player) zero-sum games, which does not require to be played in two rounds.

Brown et. al propose a similar framework, Adversarial Constraint Satisfaction Problems (Adversarial CSPs) [14], which focuses on the case where two opponents take turns to assign variables, each trying to direct the solution towards their own objectives. They describe the process as a game-tree search, and the solving algorithm is based on the minimax algorithm. Our work pushes the idea further by: 1) applying alpha-beta pruning, and 2) exploiting costs information using soft constraint techniques. We can view our work as one of their future directions.

6.3 Future Works

Our results suggest alpha-beta pruning techniques can be a future possible direction for enhancing QeCode. In Chapter 5, we can see that the $\exists$-orqset of a QCOP/Q-COP+ must have either a maximization condition, or a minimization condition, or an any condition. It may be worthwhile to apply alpha-beta prunings to solve QCOP/QCOP+ with many $\exists$-orqset having different optimization conditions.

Our pruning conditions which follow from alpha-beta prunings do not necessarily require: 1) constraint costs must be integers, and 2) constraint costs from different constraints are added together. In fact, if the cost valuation structure for constraints changed, we can still apply the sufficient pruning conditions. We are interested to achieve consistency notions obtaining the lower/upper bound based on other cost valuation structures. It may be worthwhile to define quantified VC-SPs [40]/Semiring-CSPs [10], together with consistency notions utilizing the same pruning conditions.

It will be interesting to study how to model non-zero sum optimization problems. One way to enhance our framework is to allow constraints to give costs in higher dimension integers $\mathbb{Z}_+^n$, where n is a fixed constant. We can easily model n player non-zero sum games by writing down costs for the i^{th} player in the i^{th} coordinate of the constraint costs. We then generalize quantifiers $\min_i$ ($\max_i$ resp.) to select sub-problems minimizing (maximizing resp.) the i^{th} coordinate of the costs. We will then generalize the pruning conditions, and devising consistency notions for these conditions.

Another possible future direction is to add restricted quantified set of variables $rqsets$ in QCOP+. This can make our framework to model a wider variety of applications.

In Chapter 5, we show preliminary results on value ordering heuristics. It will be interesting to investigate other variable/value ordering heuristics.

Bibliography

[1] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, USA, 2003.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 1st edition, 2009.

[3] Fahiem Bacchus and Kostas Stergiou. Solution directed backjumping for QCSP. In *CP'07*, pages 148–163, 2007.

[4] Marco Benedetti, Arnaud Lallouet, and Jeremie Vautard. QCSP made practical by virtue of restricted quantification. In *IJCAI'07*, pages 38–43, 2007.

[5] Marco Benedetti, Arnaud Lallouet, and Jeremie Vautard. Quantified constraint optimization. In *CP'08*, pages 463–477, 2008.

[6] Christian Bessière. Arc-consistency and arc-consistency again. *Artif. Intell.*, 65:179–190, January 1994.

[7] Christian Bessière. Refining the basic constraint propagation algorithm. In *IJCAI'01*, pages 309–315, 2001.

[8] Christian Bessière, Eugene C. Freuder, and Jean-Charles Regin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125 – 148, 1999.

[9] Christian Bessière, Jean-Charles Regin, Roland H.C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165 – 185, 2005.

[10] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *JACM*, 44:201–236, 1997.

[11] Lucas Bordeaux, Marco Cadoli, and Toni Mancini. CSP properties for quantified constraints: Definitions and complexity. In *AAAI'05*, pages 360–365, 2005.

[12] Lucas Bordeaux and Eric Monfroy. Beyond np: Arc-consistency for quantified constraints. In *CP'02*, pages 371–386, 2002.

[13] Jerome Bracken and James T. McGill. Mathematical programs with optimization problems in the constraints. *Operations Research*, 21(1):pp. 37–44, 1973.

[14] Kenneth N. Brown, James Little, Paidi J. Creed, and Eugene C. Freuder. Adversarial constraint satisfaction by game-tree search. In *ECAI'04*, pages 151–155, 2004.

[15] Hubert Ming Chen. *The computational complexity of quantified constraint satisfaction*. PhD thesis, Cornell University, 2004.

[16] M. C. Cooper, S. De Givry, and T. Schiex. Optimal soft arc consistency. In *IJCAI'07*, pages 68–73, 2007.

[17] M.C. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.

[18] Simon de Givry, Matthias Zytnicki, Federico Heras, and Javier Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *IJCAI'05*, pages 84–89, 2005.

[19] Ian P. Gent, Peter Nightingale, and Andrew Rowley. Encoding quantified CSPs as quantified boolean formulae. In *ECAI ’04*, pages 176–180, 2004.

[20] Ian P. Gent, Peter Nightingale, Andrew Rowley, and Kostas Stergiou. Solving quantified constraint satisfaction problems. *Artificial Intelligence*, 172(6-7):738–771, 2008.

[21] Ian P. Gent, Peter Nightingale, and Kostas Stergiou. QCSP-Solve: A solver for quantified constraint satisfaction problems. In *IJCAI’05*, pages 138–143, 2005.

[22] Javier Larrosa and Thomas Schiex. In the quest of the best form of local consistency for weighted CSP. In *IJCAI’03*, pages 239–244, 2003.

[23] Javier Larrosa and Thomas Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1 – 26, 2004.

[24] Jimmy Ho Man Lee and Ka Lun Leung. Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In *IJCAI’09*, pages 559–565, 2009.

[25] Nicolas Levasseur, Patrice Boizumault, and Samir Loudni. A value ordering heuristic for weighted CSP. In *ICTAI ’07*, pages 259–262. IEEE Computer Society, 2007.

[26] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99 – 118, 1977.

[27] Nikos Mamoulis and Kostas Stergiou. Algorithms for quantified constraint satisfaction problems. In *CP’04*, pages 752–756, 2004.

[28] K. M. Mjelde. *Methods of Allocation of Limited Resources*. John Wiley, 1983.

[29] R. Mohr and T. Henderson. Arc and path consistency revised. *Artificial Intelligence*, 28(2):225 – 233, 1986.

[30] Roger Mohr and G. Masini. Good old discrete relaxation. In *European Conference on Artificial Intelligence*, pages 651–656, 1988.

[31] Peter Nightingale. Non-binary quantified CSP: algorithms and modelling. *Constraints*, 14(4):539–581, 2009.

[32] M.J. Osborne and A. Rubinstein. *A course in game theory*. MIT Press, 1994.

[33] Wanlin Pang and Scott D. Goodwin. Characterizing tractable CSPs. In *AI '98*, pages 259–272, 1998.

[34] Mark Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53(2-3):329 – 342, 1992.

[35] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Specific filtering algorithms for over-constrained problems. In *CP'01*, pages 451–463, 2001.

[36] Cédric Pralet, Thomas Schiex, and Gérard Verfaillie. *Sequential Decision-Making Problems - Representation and Solution*. Wiley, 2009.

[37] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[38] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[39] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the Second International Workshop on CP'94*, pages 125–129, 1994.

[40] Thomas Schiex, Helene Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *IJCAI'95*, pages 631–637, 1995.

[41] David Stynes and Keeneth N. Brown. Realtime online solving of quantified CSPs. In *CP'09*, pages 771–786, 2009.

[42] David Stynes and Kenneth N. Brown. Value ordering for quantified CSPs. *Constraints*, 14(1):16–37, 2009.

[43] C. Truchet and P. Codognet. Musical constraint satisfaction problems solved with adaptive search. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 8:633–640, 2004.

[44] Guillaume Verger and Christian Bessière. Blocksolve: A bottom-up approach for solving quantified CSPs. In *CP'06*, pages 635–649, 2006.

[45] Guillaume Verger and Christian Bessière. Guiding search in QCSP+ with back-propagation. In *CP'08*, pages 175–189, 2008.

[46] H. von Stackelberg. *The Theory of the Market Economy*. Oxford University Press, 1952.

[47] Toby Walsh. Stochastic constraint programming. In *ECAI '02*, pages 111–115, 2002.

[48] G. Yu. Min-max optimization of several classical discrete optimization problems. *Journal of Optimization Theory and Applications*, 98:221–242, 1998.

[49] Yuanlin Zhang and H. C. Yap. Making ac-3 an optimal algorithm. In *IJCAI'01*, pages 316–321, 2001.