# Tractable Projection-Safe Soft Global Constraints in Weighted Constraint Satisfaction

WU, Yi

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


Thesis/Assessment Committee

Professor Young Fung Yu (Chair)

Professor Jimmy Lee Ho Man (Thesis Supervisor)

Professor Cai Leizhen (Committee Member)

Doctor Thomas Schiex (External Examiner)

# Abstract

In maintaining consistencies, such as GAC*, FDGAC* and weak EDGAC*, for soft global constraints, Weighted CSP (WCSP) solvers rely on the projection and extension operations on constraints, requiring efficient minimum cost computation. Since these operations modify the structure of the constraints, an important issue is *tractable projection-safety*, which concerns whether the minimum cost computation of a projected/extended constraint remains tractable. In this thesis, we prove that tractable projection-safety is always *possible* for projections/extensions to/from the nullary constraint ($C_\varnothing$), and always *impossible* for projections/extensions to/from $n$-ary constraints for $n \geq 2$. When $n = 1$, the answer is indefinite. We show an example that is not tractable projection-safe, while Lee and Leung give flow-based projection-safe constraints as positive examples of tractable projection-safety. We define *polynomially decomposable* soft constraints, which are amenable to tractable minimum cost computation. We further show that such constraints remains polynomially decomposable after projections/extensions to unary constraints and thus being tractable projection safe. We show that the $\texttt{soft\_among}^{var}$, $\texttt{soft\_regular}^{var}$, $\texttt{soft\_grammar}^{var}$ and `max_weight`/`min_weight` constraints are polynomially decomposable. We embed these constraints in a WCSP solver and conduct experiments to confirm the feasibility and efficiency of our proposal.

# 摘要

在對軟性約束維護一致性的過程中，例如GAC*, FDGAC*,和弱EDGAC*，加權約束滿足問題(weighted constraint satisfaction, WCSP)的求解器依賴於對約束的投影和擴展操作。由於這些操作改變了約束的結構，可解投影安全性(tractable projection-safety)就成為了一個重要的問題。一個約束是可解投影安全的，意味著即使經過了投影和擴展操作，這個約束的最小值依然是可解的。在這篇論文裡面，我們證明了零元的投影和擴展操作並不影響可解投影安全性，而二元的投影和擴展操作會使得約束失去可解投影安全性。對於一元的情形，投影和擴展操作對可解投影安全性的影響並不確定。對此我們舉出了正面和反面的例子。我們定義了一類可多項式分解(polynomially decomposable)的約束。這類約束允許我們有效地求解它們的最小值。進一步地，我們證明了即使經過了投影和擴展操作，它們的最小值依然可以被有效地求解，故而它們是可解投影安全的。我們展示了很多典型的約束都是可多項式分解的。我們在加權約束滿足問題求解器上面實現了我們提出的方法，用實驗數據顯示了我們的方法的可行性和有效性。

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This thesis reports work on *tractable projection-safe global constraints in weighted constraint satisfaction*, which is a common soft constraint framework. We address the issue of tractable projection-safety in enforcing WCSP consistencies. In this chapter, we first briefly describe *constraint satisfaction problems* (CSPs) and *weighted constraint satisfaction problems* (WCSPs). Then we give the motivation of our work, and overview the structure of rest of this thesis.

## 1.1 Constraint Satisfaction Problems

Many combinatorial problems can be model as *constraint satisfaction problems* (CSPs). As defined by Mackworth [33], a CSP is described as follows:

> *We are given a finite set of variables, a finite 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 from the domains to the variables so that all the constraints are satisfied simultaneously.*

We use the well-known $n$-queens problem to illustrate how to model a combinatorial problem as a CSP.

**Example 1.1.** *The n-queens problem is to place n queens onto a $n \times n$ chess board such that no two queens attack each other. Two queens attack each other if they share the same row, column or diagonal. Suppose $n = 4$. To model this problem, we can use four integer variables $\{x_1, x_2, x_3, x_4\}$. These variables represent the position of each queen, i.e. the i-th queen placed in the i-th row and $x_i$-th column. The following constraints are posted to restrict that no two queens attack each other:*

- *column: $x_i \neq x_j$ for all $1 \leq i < j \leq 4$, and;*
- *diagonal: $|x_i - x_j| \neq j - i$ for all $1 \leq i < j \leq 4$.*

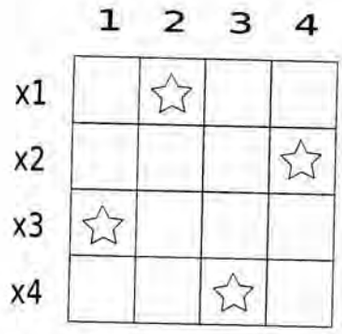


*Figure 1.1: The 4-queens problem*

*The model implicitly guarantees no two queens share the same row. Figure 1.1 shows a solution to the 4-queens problem. In fact there are two solutions in this case: $(2, 4, 1, 3)$ and $(3, 1, 4, 2)$. (We use tuples to represent an assignment, where the i-th component of the tuple corresponds to the value assigned to the i-th variable $x_i$.)*

The CSP framework is a powerful tool to model a wide range of combinatorial problems. Yet CSP is NP-Complete, which means unless P=NP, solving CSP would take exponential time in general. In practice CSPs are solved via backtracking tree search. Along a branch of a search tree, variables are assigned one by one until a solution is found or inconsistent is detected. In the later case the solver backtracks and tries another branch. To improve

just removing infeasible values in variable domains, consistency techniques in WCSP take cost information into account and retrieve hidden information by transporting costs.

## 1.3 Motivation and Goal

A global constraint is a constraint specified by its semantics, and involve a non-fixed number of variable. Besides an efficient branch and bound procedure augmented with powerful consistency algorithms, a practical WCSP solver should have a good library of soft global constraints to cater for the often complex scenarios in real-life applications. Lee and Leung [30, 31, 32] showed how AC* [18], FDAC* [27] and EDAC* [19] can be generalized and implemented efficiently for a special class of soft global constraints, namely those that are (flow-based) projection-safe[30, 32].

Lying in the heart of all WCSP consistency algorithms are (a) computation of minimum cost of constraints and (b) the projection and extension operations which transport costs among constraints to create pruning opportunities. In the case of soft global constraints which usually have high arities, specialized polynomial time algorithms can be developed for minimum cost computation according to the semantics of the global constraints and their violation measures. However, projections and extensions modify a constraint so that its structure and even semantics might change, possibly making the original minimum cost algorithm no longer applicable. Therefore, the key notions here is *tractable projection-safety*, which concerns whether the minimum cost computation of a projected/extended soft global constraint remains tractable. We discover that different consistency notions depend on different scenarios of projections and extensions. We study the impact of projections and extensions on tractability of soft global constraints, and give positive and negative examples.

Moreover, we discover that for several typical soft global constraints, we can

apply dynamic programming approach to compute their minimum costs, and the approach is still applicable after projections and extensions. We study their properties and define *polynomially decomposable* soft constraints, which can be decomposed into a tractable number of simpler constraints for (minimum) cost calculation. We show a soft global constraint of this class are tractable projection-safe.

## 1.4 Outline of the Thesis

The outline of this thesis is as follows. We give basic backgrounds on CSPs and WCSPs, including related concepts and solving techniques, in Chapter 2. Backgrounds on global constraints and constraint softening are also given in this chapter.

Chapter 3 defines tractability of a soft constraint, and addresses the issue of tractable projection-safety. We analyze tractable projection-safety by dividing the discussion into three cases of different scenarios of projections and extensions. We prove that a soft (global) constraint is always tractable projection-safe after projections/extensions to/from the nullary constraint ($C_\varnothing$), and always non-tractable after projections/extensions to/from $n$-ary constraints for $n \geq 2$. When $n = 1$, the answer is indefinite. We give a simple tractable constraint and show how it becomes non-tractable after projections/extensions to/from unary constraints, while flow-based projection-safe constraints [30, 32] are positive examples of tractable projection-safe constraints.

We also define polynomially decomposable constraints in this chapter. We define safe decomposition where a constraint is divided into sub-constraints which allows us to (1) compute the minimum cost of the original constraint from the minimum cost of its sub-constraints, and (2) distribute projections and extensions to its sub-constraints. We give special scenarios of safe decomposition. Base on safe decomposition, we define polynomially decomposable

constraints, and show that with a soft global constraint of this class, we can apply a dynamic programming approach to compute its minimum cost, and the algorithm is still applicable after projections and extensions. As such, a polynomially decomposable soft global constraint is tractable projection-safe.

Chapter 4 give examples of polynomially decomposable constraints. These constraints include the $\texttt{soft\_among}^{var}$ constraint, the $\texttt{soft\_regular}^{var}$ constraint, the $\texttt{soft\_grammar}^{var}$ constraint, and `max_weight/min_weight` constraints. For each constraint presented in this chapter, we show how they can be safely decomposed in a recursive way. Base on the decomposition, we give algorithms to calculate their minimum costs. The algorithms presented in this chapter are special cases of the generic algorithm to compute the minimum cost of a polynomially decomposable constraint.

Chapter 5 shows the experiment results. For each constraint discussed in Chapter 4, we conduct one experiment to show the efficiency of our technique. We also compare our technique with the flow-based approach by Lee and Leung [30, 32].

We conclude the thesis in Chapter 7. We summarize our work on the thesis, and shed light on possible future directions of our research.

# Chapter 2

# Background

In this chapter, we give the basic background for the rest of this thesis, including the concept of constraint satisfaction problems (CSPs), weighted constraint satisfaction problems (WCSPs), and global constraints. We also describe various consistency techniques for CSPs and WCSPs and how they are incorporated into backtracking search to build efficient solvers for these problems.

## 2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) is a tuple $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of *variables* $\{x_1, x_2, \ldots, x_n\}$. Each variable has its *domain* $D(x_i) \in \mathcal{D}$ of values that can be assigned to it. Assigning a value $v$ to a variable $x_i$ is denoted by $x_i \mapsto v$. In this thesis we assume the domains to be finite. An assignment $\{x_{s_1} \mapsto v_{s_1}, x_2 \mapsto v_{s_2}, \ldots, x_{s_{n'}} \mapsto v_{s_{n'}}\}$ on $S = \{x_{s_1}, x_{s_2}, \ldots, x_{s_{n'}}\} \subset \mathcal{X}$ can be represented as a tuple $l = (v_1, v_2, \ldots, v_n)$. The notation $l[x_{s_i}]$ denotes the value assigned to $x_{s_i}$, i.e. $v_{s_i}$, and $l[S']$ denotes the tuple formed by extracting an assignment on a subset $S' \subset S$ in $l$. We also use the notation $\mathcal{L}(S)$ to denote the set of all tuples corresponding to all possible assignments on $S = \{x_{s_1}, x_{s_2}, \ldots, x_{s_{n'}}\} \subset \mathcal{X}$, i.e. $\mathcal{L}(S) = D(x_{s_1}) \times \ldots \times D(x_{s_{n'}})$. A *hard constraint* $C_S^h \in \mathcal{C}$ over the subset of variables $S$ is a subset of $\mathcal{L}(S)$, specifying the allowed tuples to be assigned to the variables in $S$. The set of variable $S$ is the *scope* of

$C_S^h$. The constraint could be explicitly given by a table of tuples, or implicitly by its semantics. The *arity* of $C_S^h$ is defined as $|S|$. A assignment $l \in \mathcal{L}(S)$ *satisfies* $C_S^h$ if $l[S] \in C_S^h$. A *solution* of a CSP is a complete assignment that satisfies every constraint in $\mathcal{C}$. See Example 1.1 for a simple example of CSP.

The superscript $h$ in the notation $C_S^h$ is to differentiate a hard constraint from a soft constraint. Soft constraints will be discussed in Section 2.2.

To build efficient solvers for CSPs, backtracking tree search can be used. The backtracking tree search algorithm explores the whole search space in a systematic way, and backtracks as soon as it detects any failure. By examining local substructure, local consistency techniques are able to reduce search space and help the search procedure backtrack earlier.

### 2.1.1 Backtracking Tree search

Backtracking tree search is a general algorithm that systematically explores the whole search space to look for solutions of a problem. In our application, given a CSP, the whole search space is made up of all possible assignments to the variables in the CSP. The task is to find solutions of the CSP in the search space. The algorithm traverses the search space in a depth-first manner. Whenever conflict is detected, it immediately backtracks and switches to another branch. Algorithm 2.1 shows the pseudo-code for finding the first solution of a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ [4].

The algorithm starts from an empty assignment, and tries to extend it into a solution. It recursively calls `search()` to traverse the search tree. On each node of the search tree, it picks an unassigned variable $x_i$ by the function `chooseUnassignedVar()` (Line 5), and extends the assignment by assigning to it a value $v$ in its domain $D(x_i)$ by the function `chooseVal()` (Line 7-8). It then checks whether there is any conflict in the new partial assignment $l'$. If no conflicts are found, the algorithm proceeds to the sub-search tree and

repeats the procedure. The algorithm halts immediately after the first solution is found (Line 11), or the whole search tree is exhausted. In the latter case, the input CSP has no solutions.

**Algorithm 2.1**: Backtracking tree search algorithm for solving CSPs

```
1  Procedure solve() begin
2      search(∅, 𝒟) ;
3  end
4  Procedure search(l, 𝒟) begin
5      x_i ← chooseUnassignedVar() ;
6      while D(x_i) ≠ ∅ do
7          v ← chooseVal(D(x_i)) ;
8          l' ← l ∪ {x_i ↦ v} ;
9          if noConflict(l') then
10             if |l'| = |𝒳| then
11                 return l' ;
12             else
13                 sol = search(l', 𝒟) ;
14                 if sol ≠false then return sol ;
15         D(x_i) ← D(x_i) \ {v} ;
16     return false;
17 end
```

Figure 2.1 shows a search tree for Example 1.1. The variables are being assigned in the order of their indices. The algorithm starts with an empty assignment that corresponds to the empty configuration. It first tries the assignment $x_1 \mapsto 1$ (putting the first queen in the corner). No conflicts can be seen at this point. Then it proceeds to assign 1 to $x_2$. At least one of the constraints is violated due the two queens are attacking each other. So it backtracks and try another branch. The procedure continues until in the rightmost branch in the figure, when it finds a solution (which is $(2, 4, 1, 3)$). The algorithm outputs the solution and halts.

In the above example, we backtrack only when two queens are attacking each other. This strategy yields a large search tree of 27 search nodes. For more difficult problems, we may get even larger search trees. By doing more

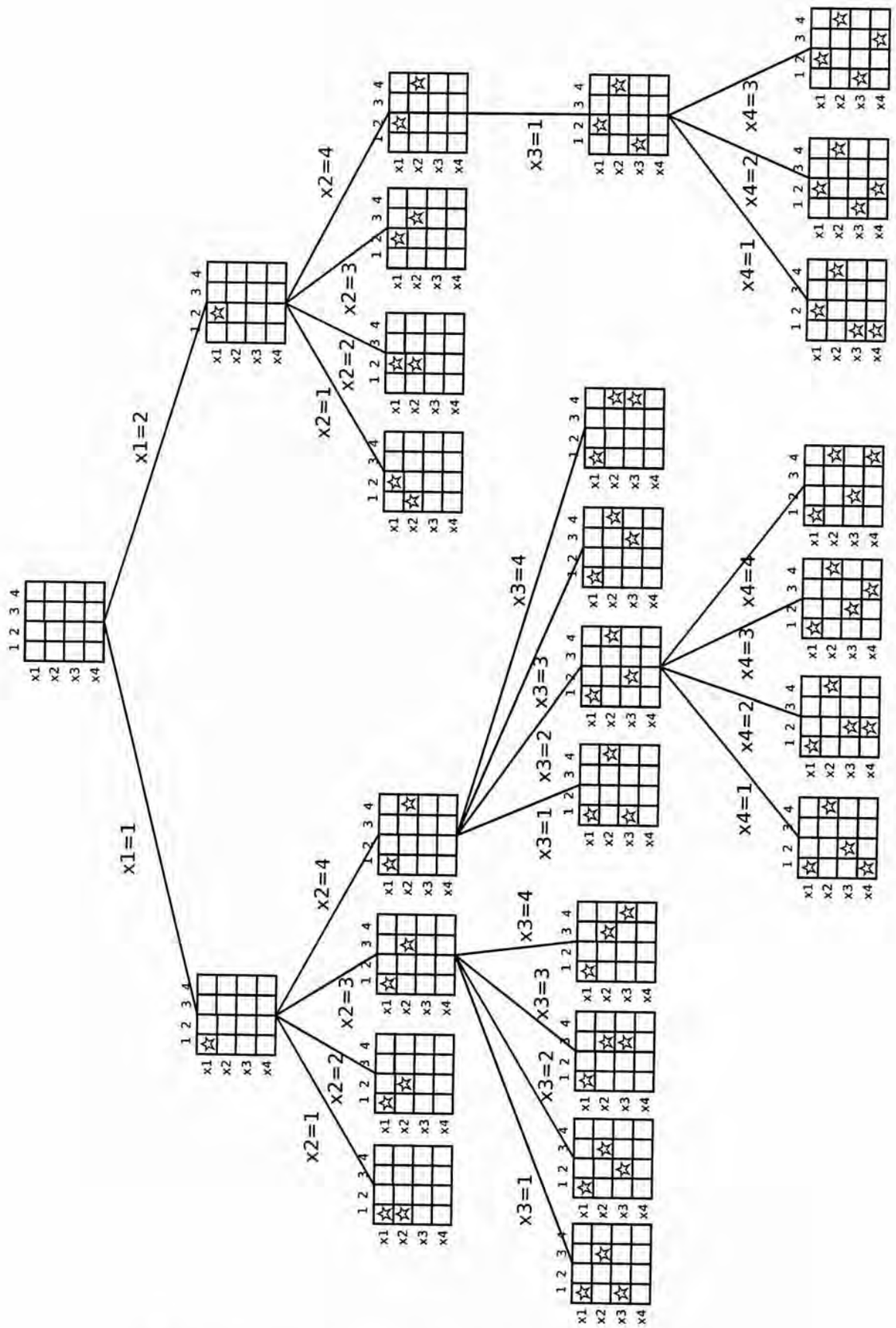


Figure 2.1: A search tree for the 4-queens problem

checking, we can prune large parts of the search tree to reduce our effort in doing search. Local consistency techniques [37, 34] are the tools to do this kind of job.

### 2.1.2 Local consistencies in CSP

Naive implementation of the backtracking search algorithm would not result in efficient solvers for CSP. One of the problems is its late detection of conflicts. For example, in the 4-queens problem, when we place the first queen in the first column, we should immediately know no other queens should be placed in the same column. So we can safely remove value 1 from the domains of $x_2$, $x_3$ and $x_4$, avoiding unnecessary search. The main idea here is to turn a CSP into another one that is equivalent but easier to solve.

**Definition 2.1.** *[4] Given two CSPs* $P_1 = (\mathcal{X}, \mathcal{D}_1, \mathcal{C}_1)$ *and* $P_2 = (\mathcal{X}, \mathcal{D}_2, \mathcal{C}_2)$. $P_1$ *is equivalent to* $P_2$ *if the have the same set of solutions.*

An *equivalence preserving transformation* converts a CSP into another equivalent CSP. Such a transformation is usually done by removing values in domains that will not appear in any solution of the CSP. Usually we are to transform a CSP into another one that of some form of *local consistency.* Different consistency notions have appeared in literature. These consistency notions give rules to filter out unwanted values in domains. Algorithms that enforce local consistencies are called constraint propagation algorithms. These algorithms look into the substructures of a CSP and turns the CSP into desired form. Two common consistency notions are *node consistency* [37, 34] and *arc consistency* [37, 34]. We are to describe them in the following.

**Node Consistency** Node consistency is perhaps the simplest form of consistency notion. It considers each time a unary constraint, that is a constraint involving one single variable.

**Definition 2.2.** *[37, 34] Given a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. A constraint $C_S^h$ is node consistent (NC) if either $|S| > 1$, or $|S| = \{x_i\}$ and for any value $v \in D(x_i)$, assigning $v$ to $x_i$ satisfies $C_S^h$. A CSP is node consistent if every constraint $C \in \mathcal{C}$ is node consistent.*

**Example 2.1.** *Suppose the domain of $x$ is $D(x) = \{2, 3, 4, 5\}$. And we have a constraint $x \leq 4$. We can remove 5 from $D(x)$ since $x \mapsto 5$ violates the constraint. Other values remains intact because assigning them to $x$ satisfies the constraint.*

**Algorithm 2.2**: Enforcing node consistency

```
1 Procedure enforceNC(X, D, C) begin
2     for C_S^h ∈ C and S = {x_i} do
3         for v ∈ D(x_i) do
4             if {x_i ↦ v} ∉ C_S^h then
5                 D(x_i) ← D(x_i) \ {v} ;
6 end
```

To enforce node consistency, we just have to check each unary constraint, and remove values that violates the constraint in the corresponding domain (Algorithm 2.2). Node consistency is simple, and yet weak in pruning power. So usually we need a stronger form of consistency notions to help discovering hidden information in a CSP.

**Arc Consistency** Arc consistency is a consistency notion that takes binary constraints (a constraint involving two variables) into account.

**Definition 2.3.** *[37, 34] Given a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. A binary constraint $C_{ij}^h \in \mathcal{C}$ over variables $x_i$ and $x_j$ is arc consistent (AC) if*

- *for every value $v_i \in D(x_i)$, these is a value $v_j \in D(x_j)$ such that $(a, b) \in C_{ij}^h$, and*

- *for every value* $v_j \in D(x_j)$*, these is a value* $v_i \in D(x_i)$ *such that* $(a, b) \in C_{ij}^h$.

*P is arc consistent if all its binary constraints are arc consistent.*

Let $v_i$ be a value in $D(x_i)$ and $v_j$ a value in $D(x_j)$. The value $v_j$ is a *support* of $v_i$ if the tuple $(v_i, v_j)$ belongs to $C_{ij}^h$. In another word, a binary constraint is arc consistent if all the values in each domain of its variables has at least a support in the other domain.

**Example 2.2.** *Consider a CSP with two variables* $\mathcal{X} = \{x_1, x_2\}$ *with domains* $D(x_1) = \{2, 3\}$ *and* $D(x_2) = \{1, 2, 3\}$*. These is only one constraint* $x_1 \times x_2 \leq 5$*. The constraint is not arc consistent. The value* 3 *in the domain of* $D(x_2)$ *has no support in* $D(x_1)$ *because even we take* $x_1 \mapsto 2$*,* $2 \times 3 = 6 > 5$*. After removing* 3 *from* $D(x_2)$*, the CSP is arc consistent.*

To enforce arc consistency, we loop through every binary constraints of a CSP, and look for supports for every value of both domains. If a value has no support, it is removed because it cannot appears in any solution of the CSP. The removed value could be the support of other values. It is thus possible that the removal causes other value to lose their support. It is necessary to repeat the process and verify that every values in the domains still has at least one support. We can stop if a fixed point is reached — no more values can be removed and every binary constraint is arc consistent. The algorithm is demonstrated in Algorithm 2.3. The algorithm is called AC-1 [34].

The algorithm can be improved. In each iteration, AC-1 tries to revise every constraint in the system even if the corresponding domains are not changed. A more efficient way is to use a queue $\mathcal{Q}$ that stores every potentially arc inconsistent constraint. Only those constraints will be revised. The algorithm terminates when the queue is empty. Such an algorithm is called AC-3 [34] and is demonstrated in Algorithm 2.4. Note that AC-1 and AC-3 differ only in the

**Algorithm 2.3**: Enforcing arc consistency (AC-1)

```
1  Procedure AC-1(X, D, C) begin
2      changed ← true ;
3      while changed = true do
4          changed ← false ;
5          foreach C_ij^h ∈ C do
6              changed←changed∨Revise(C_ij^h, x_i, x_j)∨Revise(C_ij^h, x_j, x_i) ;
7  end
8  Procedure Revise(C_ij^h, x_i, x_j) begin
9      deleted← false ;
10     foreach v_i ∈ D(x_i) do
11         if ¬∃v_j ∈ D(x_j) such that (v_i, v_j) ∈ C_ij^h then
12             D(x_i) ← D(x_i) \ {v_i} ;
13             deleted← true ;
14     return deleted;
15 end
```

main procedure. The `Revise` procedure is the same. The queue $\mathcal{Q}$ is called the *propagation queue*, since it helps propagate the consistency information from one constraint to the others.

**Algorithm 2.4**: Enforcing arc consistency (AC-3)

```
1  Procedure AC-3(X, D, C) begin
2      Q ← {C_ij^h ∈ C};
3      while Q ≠ ∅ do
4          C_ij^h ← pop(Q) ;
5          if Revise(C_ij^h, x_i, x_j) then
6              Q ← Q ∪ {C_ik^h ∈ C} ;
7          if Revise(C_ij^h, x_j, x_i) then
8              Q ← Q ∪ {C_jk^h ∈ C} ;
9  end
```

There are still rooms left for us to further improve AC-3. Various algorithms have been proposed. Examples are AC-4 [35], AC-5 [39], AC-6 [9], AC-7 [10], AC-2001 [13], AC-3.1 [50] and AC-2001\3.1 [14].

Arc consistency strikes a balance between propagation efficiency and power of removing unnecessary values. It turns out to be a practical consistency

notion and is implemented in most (if not all) CSP solvers.

**Generalized Arc Consistency** Arc consistency has been generalized to $n$-ary constraints involving $n$ variables.

**Definition 2.4.** *A constraint $C_S^h$ over a set of variables $S$ is* generalized arc consistency *(GAC) if for every value $v_i \in D(x_i)$ where $x_i \in S$, there exists a tuple $l \in \mathcal{L}(S)$ such that $l[x_i] = v_i$ and $l \in C_S^h$. A CSP is generalized arc consistent if all its constraints are generalized arc consistent.*

In another word, a constraint $C_S^h$ is arc consistent if for every variable $x_i \in S$ and every value $v_i$ in the domain of $x_i$, the assignment $\{x_i \mapsto v_i\}$ can be extended to a tuple (an assignment to the variables in $S$) $l$ that satisfies the constraint $C_S^h$. The tuple is called a *support* of $v_i \in D(x_i)$ with respect to $C_S^h$. Again unless all the constraints are GAC, a CSP cannot be GAC.

**Example 2.3.** *Consider a CSP with three variables $\mathcal{X} = \{x_1, x_2, x_3\}$ with domains $D(x_1) = \{2, 3\}$, $D(x_2) = \{1, 4\}$ and $D(x_3) = \{2, 4\}$. There is only one constraint $x_1 + x_2 + x_3 \geq 9$. The constraint is not GAC because value 1 in $D(x_2)$ has no support. For example, if we take the assignment $\{x_1 \mapsto 3, x_2 \mapsto 1, x_3 \mapsto 4\}$, it does not satisfy the constraint because $3 + 1 + 4 = 8 < 9$. The CSP is GAC after removing 1 from $D(x_2)$.*

The AC enforcement algorithms, for example, AC-3, discussed earlier in this section can be easily modified to achieve GAC.

**Combining Local Consistency with Search** We can incorporate local consistency algorithms into backtracking tree search to improve the efficiency of search. One example is the maintaining arc consistency algorithm (MAC) [46]. At each of the search tree, before choosing a value for a variable, we enforce arc consistency to reduce the domain size. As a result, many unnecessary branches are avoided. Also, if one of the domains of variables becomes

empty, we can backtrack since in this case, a conflict is found and no solutions lies in the current branch. The removal of values are undone on backtracking. The algorithm is demonstrated in Algorithm 2.5. In each search node before branching, AC is enforced by the function `enforceAC()`. It then checks whether there exists a variable with empty domain. In this case the algorithm backtracks by returning *false*. Otherwise, it continues traversing the search tree as in Algorithm 2.1. The algorithm terminates when one solution is found, or the whole search space is exhausted.

**Algorithm 2.5**: Maintaining arc consistency (MAC) search algorithm

```
1  Procedure solve() begin
2      MAC(∅, 𝒟);
3  end
4  Procedure MAC(l, 𝒟) begin
5      enforceAC() ;
6      if ∃D(x_i) ∈ 𝒟, D(x_i) = ∅ then
7          return false;
8      x_i ← chooseUnassignedVar() ;
9      while D(x_i) ≠ ∅ do
10         v ← chooseVal(D(x_i)) ;
11         l' ← l ∪ {x_i ↦ v} ;
12         if |l'| = |𝒳| then
13             return l' ;
14         else
15             sol = search(l', 𝒟) ;
16             if sol ≠false then return sol ;
17         D(x_i) ← D(x_i) \ {v} ;
18     return false;
19 end
```

Figure 2.2 gives the search tree for the 4-queens problem. Here we search for all the solutions for this problem. The values removed by enforcing AC is marked by shaded grid. Note that in the leftmost search node where 1 is assigned to $x_1$, the search tree beneath it disappears comparing to Figure 2.1, thanks to the earlier detection of failure.

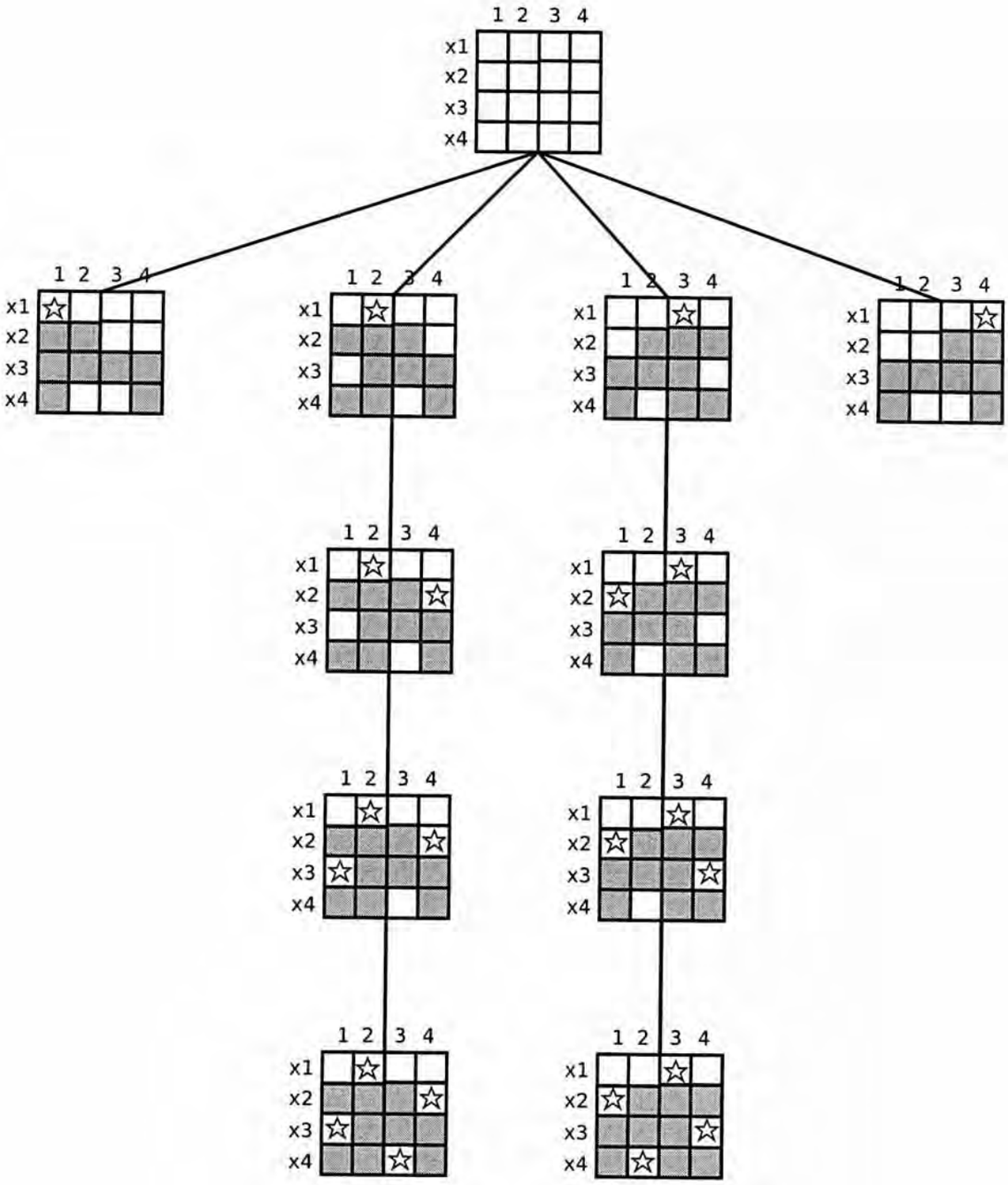


Figure 2.2: The MAC search tree for the 4-queens problem

## 2.2 Weighted Constraint Satisfaction Problems

In real-life there are optimization problems and over-constrained problems. For example, we may want to maximize our profit, or minimum the consumed resources. *Weighted constraint satisfaction* (WCSP) [48] is one of the soft constraint frameworks to handle optimization problems and over-constrained problems. Instead of being a set of allowed tuples, a constraint in WCSP is a cost function. And the total cost is the sum of costs returned by all the constraints. The task is to find a tuple to minimize the overall cost.

A *weighted CSP* (WCSP) [48] is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$. $\mathcal{X}$ is a set of *variables* $\{x_1, x_2, \ldots, x_n\}$. Each variable has its finite *domain* $D(x_i) \in \mathcal{D}$ of values that can be assigned to it. Assigning a value $v$ to a variable $x_i$ is denoted by $x_i \mapsto v$. A *tuple* $l = (v_1, \ldots, v_{n'})$ is used to represent an assignment to a set of variables $\{x_1 \mapsto v_1, \ldots, x_{n'} \mapsto v_{n'}\}$. We denote $l[x_i]$ as the value assigned to $x_i$, and $l[S]$ as the tuple formed from the assignment on variables in the set $S$. We use the notation $\mathcal{L}(S)$ to denote the set of all tuples corresponding to all possible assignments on $S = \{x_{s_1}, \ldots, x_{s_{n'}}\}$, i.e. $\mathcal{L}(S) = D(x_{s_1}) \times \ldots \times D(x_{s_{n'}})$. $\mathcal{C}$ is a set of *soft constraints*. Each constraint $C_S \in \mathcal{C}$ over a set of variable $S \subseteq X$ is a cost function which maps $l[S]$ to a value in the valuation structure $V(\top) = ([0, \ldots, \top], \oplus, \leq)$. The *scope* of a constraint $C_S$ is $S$ and its *arity* is $|S|$. The valuation structure contains a set of integers $[0, \ldots, \top]$ with standard integer ordering $\leq$. Addition $\oplus$ is defined by $a \oplus b = \min(\top, a + b)$. The subtraction $\ominus$ in $V(\top)$ is defined as

$$a \ominus b = \begin{cases} a - b, & \text{if } a < \top \\ \top, & \text{if } a = \top \end{cases}$$

$S$ is the *scope* of $C_S$. The *cost* of a tuple $l$ in a WCSP corresponding to an assignment on $\mathcal{X}$ is defined as:

$$cost(l) = \bigoplus_{C_S \in C} C_S(l[S])$$

The goal is to find an assignment on $\mathcal{X}$ with the minimum cost among all possible assignments. Such an assignment is a *solution* of the WCSP.

Without loss of generality, we assume there always exists a constraint $C_\varnothing$ over an empty set of variable, and for every variables $x_i$ in $\mathcal{X}$, a unary constraint $C_i$ over $x_i$. The constraint $C_\varnothing$ provides a lower bound of the minimum cost. We also use $C_{ij}$ to denote a binary constraint with scope $\{x_i, x_j\}$. In the rest of this thesis we refer to soft constraints as constraints.

**Example 2.4.** *Figure 2.3 shows an example of WCSP. There are two variable $x_1$ and $x_2$ in $\mathcal{X}$. The domain of $x_1$ is $D(x_1) = \{a, b\}$ and the domain of $x_2$ is $D(x_2) = \{a, b, c\}$. There are three constraints $C_1$, $C_2$ and $C_{12}$ given as tables. The upper bound $\top$ is set to* 4.

| $x_1$ | $C_1$ |
|---|---|
| $a$ | 1 |
| $b$ | 0 |

| $x_2$ | $C_2$ |
|---|---|
| $a$ | 1 |
| $b$ | 0 |
| $c$ | 2 |

| $x_1$ | $x_2$ | $C_{12}$ |
|---|---|---|
| $a$ | $a$ | 0 |
| $a$ | $b$ | 1 |
| $a$ | $c$ | 0 |
| $b$ | $a$ | 2 |
| $b$ | $b$ | 1 |
| $b$ | $c$ | 0 |

*Figure 2.3: A WCSP with three constraints*

*Figure 2.4 gives a graphical representation of the above example. A rectangle represents a variable domain. Circles represent values in domains. Numbers in the circles stand for unary costs. An edge between two circles represents a cost if the two values are taken simultaneously. A label on the edge gives the cost. If the cost is* 1*, the label is omit.*

*The solution of this WCSP is $(b, b)$ (or equivalently $\{x_1 \mapsto b, x_2 \mapsto b\}$). It has the minimum cost* 1*. There is only one solution since other tuples incur a cost greater or equal to* 2*.*

Note that a CSP is a special WCSP with $\top = 1$. Every hard constraint can be translate as a soft constraint by assigning a cost $\top$ to disallowed tuples.

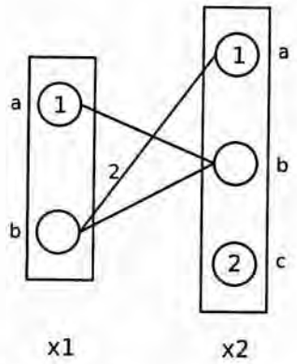


*Figure 2.4: Graphical representation of a WCSP*

In the rest of the thesis we may use the graphical representation shown in the above example to depict WCSPs with only unary and binary constraints.

### 2.2.1 Branch and Bound Search

*Branch and bound* (BnB) search [26] is a special kind of backtracking tree search. It is a general method to obtain an optimal solution for an optimization problem. Suppose we are solving a minimization problem. The algorithm traverse the whole search space as backtracking tree search. During search, a currently best solution is kept. We use it as an upper bound of the optimal solution. Initially it is set to $\top$, and is updated when a better solution is found. On each search node, the algorithm try its best to evaluate a lower bound of the cost in the current branch. If the lower bound is no less than the upper bound, it is a signal that the optimal solution cannot appears in the search tree beneath this search node. In this case the algorithm immediately backtracks. Unlike solving decision problems, where the algorithm immediately stops when it encounters the first solution (if only one solution is of interest), branch and bound search has to exhaust the whole search space to prove the currently best solution is indeed the optimal solution.

Algorithm 2.6 shows the pseudo-code of solving a WCSP with branch and bound search. During search, the upper bound $\top$ is always set to the cost of the currently best solution. To have an estimation of the lower bound, the

algorithm first transform the WCSP in a desired form by enforcing local consistency. (WCSP local consistencies will be discussed in the next subsection.) Then $C_\varnothing$ is then used as a lower bound of the optimum. The algorithm backtracks whenever a better solution is found (Line 2) or the lower bound is no less than the upper bound (Line 4).

**Algorithm 2.6**: Solving WCSP using branch and bound

```
1  Procedure BranchAndBound(X, D, C, ⊤, l) begin
2      if X = ∅ then return C_∅;
3      enforceLocalConsistency();
4      if C_∅ ≥ ⊤ then return ⊤;
5      x_i ← chooseVar(X) ;
6      foreach v ∈ D(x_i) do
7          l' ← l ∪ {x_i ↦ v} ;
8          C_∅ ← C_∅ ⊕ C_i(v);
9          ⊤ ← BranchAndBound(X, D, C, ⊤, l') ;
10     return ⊤;
11 end
```

Figure 2.5 shows a search tree for solving the WCSP in Example 2.4 using branch and bound search.

### 2.2.2 Local Consistencies in WCSP

As in solving CSPs, we can incorporate local consistency techniques with the basic branch and bound search. Local consistencies in WCSP are capable of removing infeasible values in the domains, as well as deducing a lower bound of the minimum cost. The lower bound is then used in the branch and bound search to decide whether it can immediately backtrack from the current branch. As for CSPs, consistency notions for WCSPs are achieved via equivalence preserving transformation.

**Definition 2.5.** *[18] Given two WCSPs* $P_1 = (\mathcal{X}, \mathcal{D}_1, \mathcal{C}_1, \top)$ *and* $P_2 = (\mathcal{X}, \mathcal{D}_2, \mathcal{C}_2, \top)$. $P_1$ *is equivalent to* $P_2$ *if for all feasible tuples* $l \in \mathcal{L}(\mathcal{X})$ *in both problems,* $cost_{P_1}(l) = cost_{P_2}(l)$.

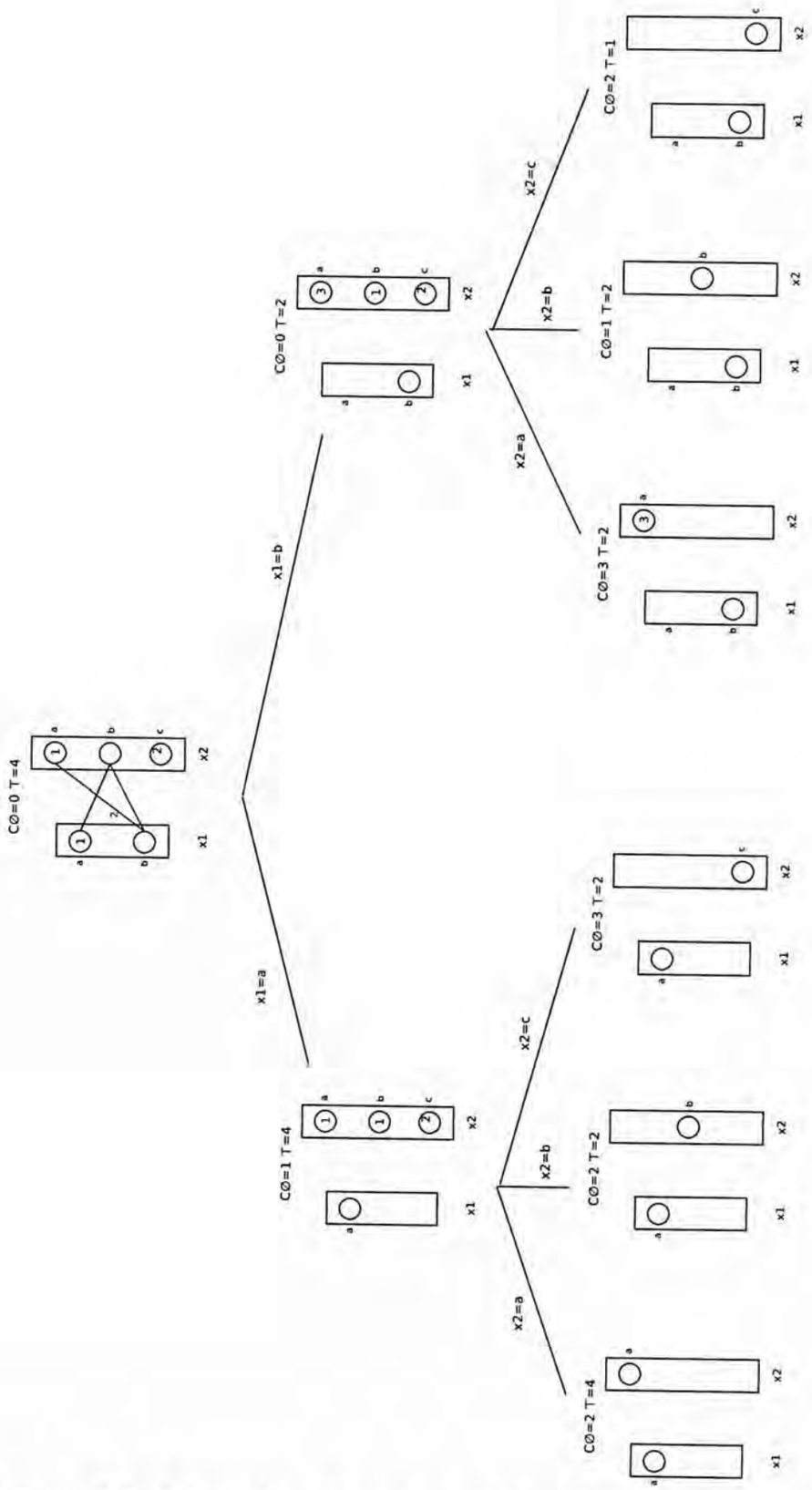


Figure 2.5: A branch and bound search tree to solve a WCSP

Note that given a WCSP, a tuple $l$ is feasible iff $cost(l) < \top$. Typical equivalence preserving transformation in WCSP are projections and extensions. We adopt the definition from Cooper [16] and Larrosa and Schiex [27]. Given $S_2 \subset S_1$ and a tuple $l$ on $S_2$. An *r-projection* of a cost $\alpha$ from $C_{S_1}$ to $C_{S_2}(l)$, where $l$ is a tuple on $S_2$ with $|S_2| = r$, is a transformation of $(C_{S_1}, C_{S_2})$ to $(C'_{S_1}, C'_{S_2})$ such that for all assignments $\hat{l}$:

$$C'_{S_1}(\hat{l}[S_1]) = \begin{cases} C_{S_1}(\hat{l}[S_1]) \ominus \alpha, & \text{if } \hat{l}[S_2] = l \\ C_{S_1}(\hat{l}[S_1]), & \text{otherwise} \end{cases}$$
$$C'_{S_2}(\hat{l}[S_2]) = \begin{cases} C_{S_2}(\hat{l}[S_2]) \oplus \alpha, & \text{if } \hat{l}[S_2] = l \\ C_{S_2}(\hat{l}[S_2]), & \text{otherwise} \end{cases}$$

An *r-extension* of a cost $\alpha$ from $C_{S_2}(l)$ to $C_{S_1}$, where $l$ is a tuple on $S_2$ with $|S_2| = r$, and $\alpha \leq C_{S_2}(l)$, is a transformation of $(C_{S_1}, C_{S_2})$ to $(C'_{S_1}, C'_{S_2})$ such that for all assignments $\hat{l}$:

$$C'_{S_1}(\hat{l}[S_1]) = \begin{cases} C_{S_1}(\hat{l}[S_1]) \oplus \alpha, & \text{if } \hat{l}[S_2] = l \\ C_{S_1}(\hat{l}[S_1]), & \text{otherwise} \end{cases}$$
$$C'_{S_2}(\hat{l}[S_2]) = \begin{cases} C_{S_2}(\hat{l}[S_2]) \ominus \alpha, & \text{if } \hat{l}[S_2] = l \\ C_{S_2}(\hat{l}[S_2]), & \text{otherwise} \end{cases}$$

In addition, if $S_2 = \emptyset$, the projections/extensions are always to/from the nullary constraint $C_\varnothing$. We note that extension is the inverse of projection if no intermediate result is $\top$.

We use $\delta$ to denote a projection to a constraint $C_{S_2}$ or an extension from a constraint $C_{S_2}$. We also use $\Delta$ to denote a series of projections/extensions. The constraint obtained by applying $\delta$ ($\Delta$) to a constraint $C_{S_1}$ is denoted by $\delta(C_{S_1})$ ($\Delta(C_{S_1})$, respectively). For convenience, when $S_1 \cap S_2 = \varnothing$, we define $\delta(C_{S_1}) = C_{S_1}$.

In the following we briefly discuss four consistency notions in WCSPs, namely NC* [28], (strong) ∅IC [51, 30], (G)AC* [28, 30], FD(G)AC* [27, 30], ED(G)AC* [19, 31], and $k$-consistency [16].

### Node Consistency Star

**Definition 2.6.** *[28] Given a WCSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$.

- *A value* $v \in D(x_i)$ *where* $x_i \in \mathcal{X}$ *is node consistent star (NC*) if* $C_\varnothing \oplus C_i(v) < \top$.
- *A variable* $x_i \in \mathcal{X}$ *is NC* if all values in* $D(x_i)$ *is NC* and there exists a value* $v \in D(x_i)$ *such that* $C_i(v) = 0$. *Such a value is called a* unary support *of* $x_i$.
- *P is NC* if all its variables are NC*.*

Note that NC* collapses to NC when a WCSP represents a CSP, i.e. $\top = 1$.

**Example 2.5.** *Consider the example given in Figure 2.6 and suppose* $\top = 4$. *The WCSP in Figure 2.6(a) is not NC*, since the unary cost of both values in* $D(x_i)$ *is larger than* 0. *At most* 1 *cost can be projected to* $C_\varnothing$ *(Figure 2.6(b)). It is still not NC* because* $C_2(c) \oplus C_\varnothing = 3 \oplus 1 = \top$. *We immediately know* $x_2 \mapsto c$ *cannot appears in any solutions. By removing the value c from* $D(x_2)$ *the WCSP is NC* (Figure 2.6(c)).*

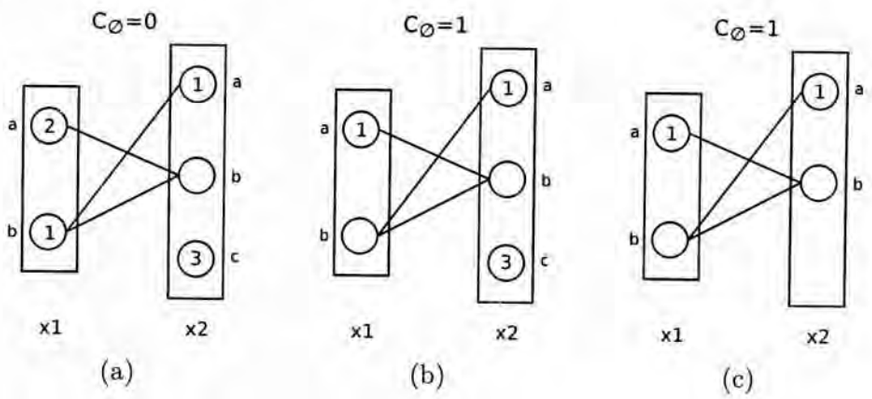


*Figure 2.6: Node consistency star*

### ∅-Inverse Consistency

**Definition 2.7.** *[51, 30] Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$.*

- *A constraint $C_S \in \mathcal{C}$ is* ∅-inverse consistent (∅IC) *if there exists a tuple $l \in \mathcal{L}(S)$ such that $C_S(l) = 0$.*
- *$C_S$ is* strong ∅-inverse consistent (strong ∅IC) *if $C_S$ is ∅IC, and for all $v \in D(x_i)$ where $x_i \in S$, there exists a tuple $l \in \mathcal{S}$ such that $l[x_i] = v$ and $C_\emptyset \oplus C_i(v) \oplus C_S(l) < \top$. Such a tuple is called the* ∅-support *of the value $v \in D(x_i)$ with respect to $C_S$.*
- *$P$ is ∅IC (strong ∅IC) if all constraints in $\mathcal{C}$ are ∅IC (strong ∅IC, respectively).*

**Example 2.6.** *Consider the example given in Figure 2.7 and suppose $\top = 4$. The binary constraint $C_{12}$ in the WCSP shown in Figure 2.7(a) is not ∅IC. By projecting cost 1 from $C_{12}$ to $C_\emptyset$, ∅IC is achieved (Figure 2.7(b)). It is not strong ∅IC, since for value a in $D(x_2)$, $C_\emptyset \oplus C_2(a) = 1 \oplus 3 = \top$. By removing a from $D(x_2)$, it is strong ∅IC (Figure 2.7(c)).*

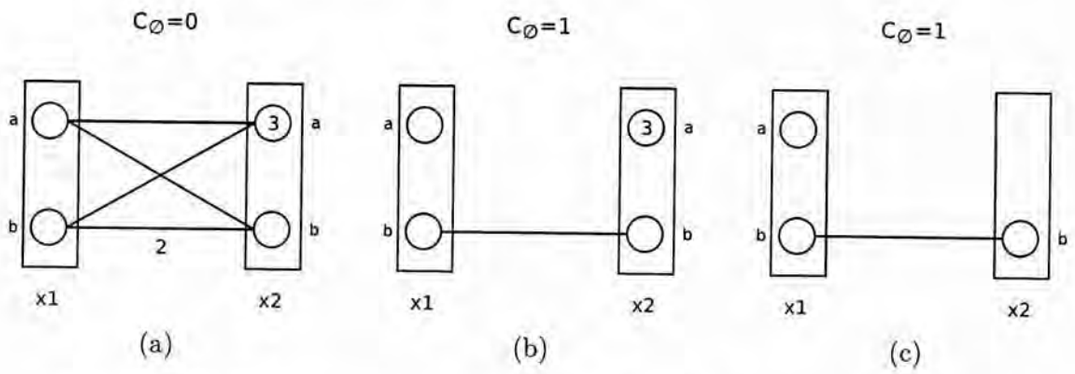


*Figure 2.7: ∅-inverse consistency and strong ∅-inverse consistency*

### (Generalized) Arc Consistency Star

**Definition 2.8.** *[28] Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$.*

- *A value $v \in D(x_i)$ where $x_i \in \mathcal{X}$ is* arc consistent star (AC*) *with respect to a binary constraint $C_{ij}$ over variables $x_i$ and $x_j$ if there exists a value $u \in D(x_j)$ such that $C_{ij}(a, b) = 0$. Such a value is called a* simple support *of $a \in D(x_i)$.*

- *A variable $x_i \in \mathcal{X}$ is AC* if it is NC* and each value in $D(x_i)$ is AC* with respect to every binary constraint over $x_i$.*

- *$P$ is AC* if all its variables are AC*.*

**Definition 2.9.** *[18] Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$. Assume $x_i \in \mathcal{X}$, $S \subseteq \mathcal{X}$ and $C_S \in \mathcal{C}$.*

- *A* simple support *of a value $v \in D(x_i)$ with respect to a soft constraint $C_S$ is a tuple $l \in \mathcal{L}(S)$ with $l[x_i] = v$ satisfying $C_S(l) = 0$.*

- *A variable $x_i \in \mathcal{X}$ is* generalized arc consistent star (GAC*) *with respect to $C_S$ if it is NC*, and each value $v \in D(x_i)$ has a simple support with respect to $C_S$.*

- *$P$ is GAC* if it is NC* and each variable is GAC* with respect to all constraints in $\mathcal{C}$.*

**Example 2.7.** *Consider the example given in Figure 2.8. The WCSP in Figure 2.8(a) is not (G)AC*, since value a in $D(x_2)$ has no support. By projecting a cost 1 to $C_2(b)$, both values in $D(x_1)$ are supports of value b in $D(x_2)$ (Figure 2.8(b)). Still, tt is not (G)AC* since now $x_2$ lost its unary support. The unary cost of $x_2$ can be projection to $C_\varnothing$ and (G)AC* is achieved (Figure 2.8(c)).*

### Full Directional (Generalized) Arc Consistency Star

**Definition 2.10.** *[27] Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$.*

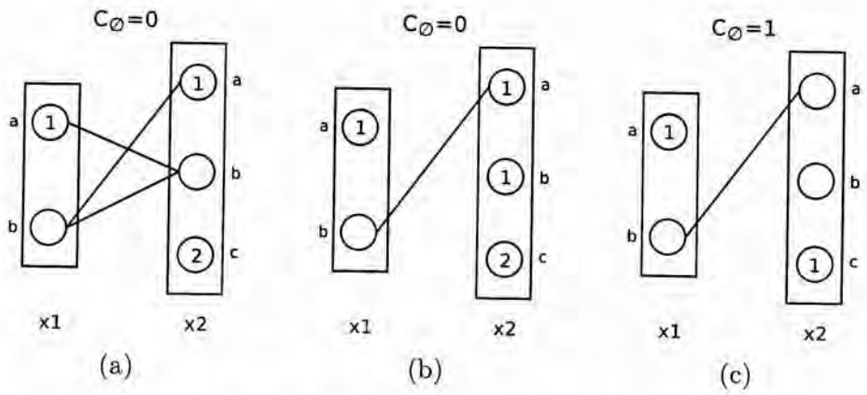


*Figure 2.8: Arc consistency star*

- *The value* $b \in D(x_j)$ *is a* full support *of a value* $a \in D(x_i)$ *if* $C_{ij}(a, b) \oplus C_j(b) = 0$.

- *The value* $a \in D(x_i)$ *is* directional arc consistent *with respect to a binary constraint* $C_{ij}$ *where* $j > i$ *if there exists a full support in* $D(x_j)$.

- *A variable* $x_i$ *is* directional arc consistent star (DAC*) *if it is NC* and each value in its domain is directional arc consistent with respect to all binary constraints* $C_{ij}$ *where* $j > i$.

- $P$ *is* fully directional arc consistent (FDAC*) *if all variables are AC* and DAC*.*

**Definition 2.11.** *[30, 32] Given a WCSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$. *Assume* $x_i \in \mathcal{X}$, $S \subseteq \mathcal{X}$ *and* $C_S \in \mathcal{C}$.

- *A* full support *of a value* $v \in D(x_i)$ *with respect to a constraint* $C_S$ *and a set of variables* $T \subseteq S \setminus \{x_i\}$ *is a tuple* $l \in \mathcal{L}(S)$ *with* $l[x_i] = v$ *such that* $C_S(l) \oplus \bigoplus_{x_j \in T} C_j(l[x_j]) = 0$.

- $x_i$ *is* directional generalized arc consistent star (DGAC*) *with respect to* $C_S$ *if it is NC* and each value in* $D(x_i)$ *has a full support with respect to* $C_S$ *and* $\{x_j | j > i\} \cap S$.

- *P is* fully directional generalized arc consistent star (FDGAC*) *if it is GAC* and each variable is DGAC* with respect to all constraints in C.*

**Example 2.8.** *Consider the example given in Figure 2.9. Value a in* $D(x_1)$ *has a full support, which is value b in* $D(x_2)$ *since* $C_{12}(a,b) \oplus C_2(b) = 0$. *Value b in* $D(x_1)$ *has no support. To transform the WCSP into a FD(G)AC* one, we can extend a cost* 1 *from* $C_2(c)$ *to the binary constraint (Figure 2.9(b) then projection a cost* 1 *from the binary constraint to* $C_1(b)$ *(Figure 2.9(c)).*

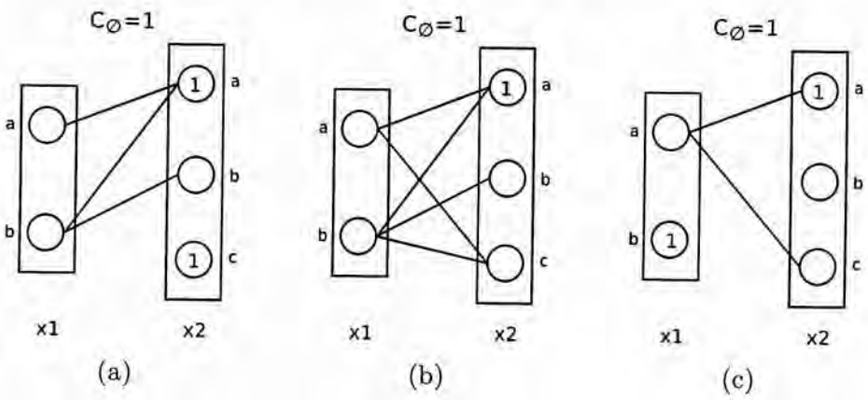


*Figure 2.9: Full directional arc consistency star*

### Existential Directional (Generalized) Arc Consistency

**Definition 2.12.** *[19] Given a WCSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$.

- *A variable* $x_i$ *is* existential arc consistent star (EAC*) *if there exists at least one value* $v \in D(x_i)$ *such that* $C_i(v) = 0$ *and it has a full support with respect to every binary constraint* $C_{ij}$. *Such a value v is called the* fully supported value *of* $x_i$.

- *P is EAC* if every variables are NC* and EAC*.*

- *P is* existential directional arc consistent star (EDAC*) *if it is FDAC* and EAC*.*

Lee and Leung [31, 32] showed that naively generalizing EDAC* to high arity constraints is not always enforceable, i.e. the algorithm may not terminate. The gave a weak form of EDGAC* base on fully support set.

**Definition 2.13.** *The* fully supported set $U(C_S, x_i)$ *for a variable* $x_i$ *and a constraint* $C_S$ *with* $x_i \in S$ *is a set of variables such that:*

- $U(C_S, x_i) \subseteq S$;
- $U(C_S, x_i) \cap U(C_k, x_i) = \varnothing$ *for two different constraints* $C_{S_j}, C_{S_k} \in \mathcal{C}$, *and;*
- $\bigcup_{C_S \in \mathcal{C} \wedge x_i \in S} U(C_S, x_i) = (\bigcup_{C_S \in \mathcal{C} \wedge x_i \in S} S) \setminus \{x_i\}$.

**Definition 2.14.** *[30, 32] Given a WCSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$ *and any fully supported set* $U(C_{ij}^m, x_i)$ *for each variable* $x_i \in \mathcal{X}$ *and each constraint* $C_{ij}^m \in \mathcal{C}$.

- *A* weak fully supported value $v \in D(x_i)$ *of a variable* $x_i \in \mathcal{X}$ *is the value with* $C_i(v) =$ *and full supports with respect to all constraints* $C_S \in \mathcal{C}$ *with* $x_i \in S$ *and* $U(C_S, x_i)$, *i.e. for every non-unary constraint* $C_S \in \mathcal{C}$, *there exists a tuple* $l \in \mathcal{L}(S)$ *with* $l[x_i] = v$ *such that* $C_S(l) \oplus \bigoplus_{x_j \in U(C_S, x_i)} C_j(l[x_j]) = 0$.
- *A variable* $x_i$ *is* weak existential generalized arc consistent star (weak EGAC*) *if it is NC* and there exists at least one weak fully supported value in its domain* $D(x_i)$.
- $P$ *is* weak existential directional generalized arc consistent star (weak EDGAC*) *if it is FDGAC* and each varible is EGAC*.*

**Example 2.9.** *Consider the example given in Figure 2.10. The WCSP in Figure 2.10(a) has three variables* $x_1, x_2, x_3$ *and two binary constraints* $C_{13}, C_{23}$. *It is not ED(G)AC.* $C_3(a) = 1 > 0$. *Value b in* $D(x_3)$ *has no full support with respect to constraint* $C_{13}$. *Value c in* $D_(x_3)$ *has no full support with respect*

*to constraint $C_{23}$. To transform the WCSP into a ED(G)AC* one, we first extend from $C_1(b)$ to $C_{13}$ and extend from $C_2(b)$ to $C_{23}$ (Figure 2.10(b)). Then, by projecting from $C_{13}$ to $C_3(b)$ and from $C_{23}$ to $C_3(c)$, both value b and c in $D(x_3)$ has at least one full support with respect to every binary constraint (Figure 2.10(c)). Finally, by a projection of cost 1 from $C_3$ to $C_\varnothing$, EDAC* is achieved.*

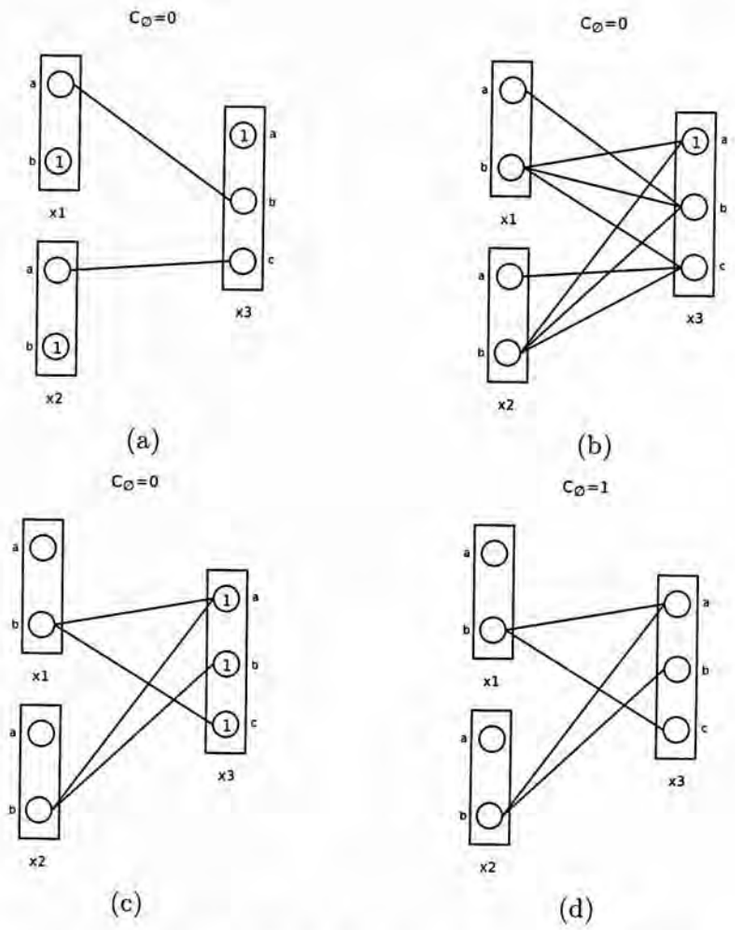


*Figure 2.10: Existential directional arc consistency star*

**$k$-Consistency** The following definition is adopted from Cooper's definition of $k$-consistency for valued constraint satisfaction problem [16].

**Definition 2.15.** *Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \top)$. Assume $S \subseteq \mathcal{X}$ and $C_S \in \mathcal{C}$. $k$ is a positive integer. $S'$ is a proper subset of $S$ where $|S'| = k$.*

- *$S'$ is k-consistent with respect to $C_S$ if for all tuples $l' \in \mathcal{L}(S')$, there is tuple $l \in \mathcal{L}(S)$ with $l[S'] = l'$ such that $C_S(l) = 0$.*
- *$P$ is k-consistent if for all subset $S' \subset \mathcal{X}$ with $|S'| = k$ is k-consistent with respect to all constraints in $\mathcal{C}$.*

**Example 2.10.** *Consider the example given in Figure 2.11. $C_S$ is a ternary constraint with the scope $S = \{x_1, x_2, x_3\}$ and $C_{12}$ is a binary constraint over $x_1$ and $x_2$. In Figure 2.11(a), $\{x_1 \mapsto a, x_2 \mapsto b\}$ is not 2-consistent with respect to $C_S$. By projecting cost 1 from $C_S$ to $C_{12}(a, b)$, it is 2-consistent (Figure 2.11(b)).*

| $x_1$ | $x_2$ | $x_3$ | $C_S$ |
|---|---|---|---|
| $a$ | $a$ | $a$ | 0 |
| $a$ | $a$ | $b$ | 0 |
| $a$ | $b$ | $a$ | 2 |
| $a$ | $b$ | $b$ | 1 |
| $b$ | $a$ | $a$ | 1 |
| $b$ | $a$ | $b$ | 0 |
| $b$ | $b$ | $a$ | 0 |
| $b$ | $b$ | $b$ | 1 |

| $x_1$ | $x_2$ | $C_{12}$ |
|---|---|---|
| $a$ | $a$ | 0 |
| $a$ | $b$ | 0 |
| $b$ | $a$ | 0 |
| $b$ | $b$ | 0 |

(a)

| $x_1$ | $x_2$ | $x_3$ | $C_S$ |
|---|---|---|---|
| $a$ | $a$ | $a$ | 0 |
| $a$ | $a$ | $b$ | 0 |
| $a$ | $b$ | $a$ | **1** |
| $a$ | $b$ | $b$ | **0** |
| $b$ | $a$ | $a$ | 1 |
| $b$ | $a$ | $b$ | 0 |
| $b$ | $b$ | $a$ | 0 |
| $b$ | $b$ | $b$ | 1 |

| $x_1$ | $x_2$ | $C_{12}$ |
|---|---|---|
| $a$ | $a$ | 0 |
| $a$ | $b$ | **1** |
| $b$ | $a$ | 0 |
| $b$ | $b$ | 0 |

(b)

*Figure 2.11: k-consistency*

## 2.3 Global Constraints

In general, in CSPs, every constraint can be represented as a table. Each entry of the table specifies whether a tuple is accepted by the corresponding constraint. Such an representation loses the semantics of the constraint. Also, the size of the table is usually exponential in the number of its variables. Thus, they are useful only for constraints involving a few variables or small domains.

In contrast to table constraints, a global constraint is a constraint specified by its semantics, and it involves a non-fixed number of variables. For example,

instead of listing out all the allowed tuples, we can post an $\texttt{allDifferent}(x_1, x_2, x_3)$ constraint, requiring $x_1$, $x_2$ and $x_3$ to take distinct values. Global constraints play an important role in the CSP framework. Many real-life problems can be easily modeled by global constraints. More over, in solving CSPs, global constraints are more efficient than table constraints due to its compact representation size and efficient consistency enforcement algorithms.

Another benefit of using global constraints is that they usually capture the semantics of a conjunction of smaller constraints. Thus, enforcing consistencies on global constraints would usually result in pruning more infeasible values.

**Example 2.11.** *Suppose in a CSP we have three variables $x_1, x_2, x_3$. $D(x_1) = D(x_2) = \{a, b\}$. $D(x_3) = \{a, b, c\}$. There are three constraints $x_1 \neq x_2$, $x_1 \neq x_3$ and $x_2 \neq x_3$. It is AC since every variable is AC with respect to every involved constraints. No values are pruned in this case. If the constraints is replaced by one* $\texttt{allDifferent}(x_1, x_2, x_3)$ *constraint, the CSP is not AC. Value $a, b \in D(x_3)$ has no support with respect to the* `allDifferent` *constraint. Thus, by enforcing AC, these two values are removed from $D(x_3)$.*

Enforcing GAC on global constraint is NP-Hard in general [12]. For specific global constraints, polynomial time algorithms to enforce AC and other consistencies have been discussed in the literature [6].

The following constraints will be discuss later in this thesis. All of the constraints below are hard constraints.

**Definition 2.16.** *(*`2sat` *constraint) Let $X$ be a set of boolean variable and $F$ a set of binary clauses. The* $\texttt{2sat}(X, F)$ *constraint requires that all the clauses in $F$ are satisfied.*

**Example 2.12.** *Suppose $X = \{x_1, x_2, x_3\}$ is a set of boolean variables. $F = \{x_1 \vee \bar{x}_2, x_2 \vee \bar{x}_3, \bar{x}_1 \vee \bar{x}_3\}$. The tuple $(true, true, false)$ satisfies the constraint since all the clauses in $F$ are satisfied.*

**Definition 2.17.** *(*`among` *constraint) [7] Let $X$ be a set of variables, $V$ a set of values. $lb$ and $ub$ are two integers satisfying $lb \leq ub$. The constraint* $\texttt{among}(X, lb, ub, V)$ *requires the number of variables taking a value in $V$ must be no less than $lb$ and no more than $ub$.*

**Example 2.13.** *Suppose $X = \{x_1, x_2, x_3\}$, $D(x_1) = D(x_2) = D(x_3) = \{a, b, c\}$, and $V = \{b, c\}$. The tuple $(c, a, b)$ satisfies the constraint* $\texttt{among}(X, 1, 2, V)$ *since the number of variables taking a value in $V$ is 2.*

**Definition 2.18.** *(*`regular` *constraint) [40] Let $M = (Q, \Sigma, T, q_0, F)$ denote a deterministic finite automaton where $Q$ is a finite set of states, $\Sigma$ is an alphabet, $T$ is a set of transitions of the form $(q_i, c) \mapsto q_j$ with $q_i, q_j \in Q$ and $c \in \Sigma$. $q_0$ is the initial state and $F \subseteq Q$ is the set of final states. $X = [x_1, \ldots, x_n]$ is a sequence of variables with domain $D(x_i) \subseteq \Sigma$. The constraint* $\texttt{regular}(X, M)$ *requires the sequence form by $X$ must belongs to the regular language recognized by $M$.*

**Example 2.14.** *Let $M$ be the deterministic finite state automaton shown in Figure 2.12. $X = [x_1, x_2, x_3]$ where $D(x_i) = \{a, b\}$. Both tuples $(a, a, b)$ and $(b, a, a)$ satisfy the constraint* $\texttt{regular}(X, M)$ *since the corresponding sequences are recognizable by $M$.*

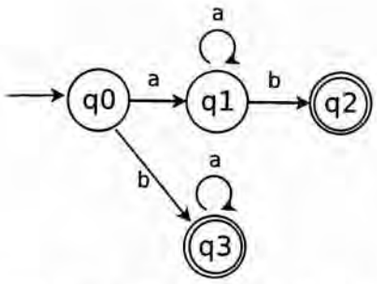


*Figure 2.12: A finite state automaton for a* `regular` *constraint*

**Definition 2.19.** *(*`grammar` *constraint) [22, 42] Let $G = (\Sigma, N, P, T)$ denote a context-free grammar where $\Sigma$ is an alphabet, $N$ is a finite set of non-terminals, $P$ is a set of productions, and $T \in N$ is the start non-terminal. We assume $G$*

*is in Chomsky Normal Form, i.e. productions in $P$ are in the form of either $(A \mapsto a)$ or $(A \mapsto BC$, where $A, B, C \in N$ and $a \in \Sigma$. $X = [x_1, \ldots, x_n]$ is a sequence of variables with domain $D(x_i) \subseteq \Sigma$. The constraint* $\texttt{grammar}(X, G)$ *requires that the sequence formed by $X$ belongs to the context-free grammar $G$.*

**Example 2.15.** *Let $G = (\Sigma, N, P, S)$ be a context-free grammar in Chomsky Normal Form where $\Sigma = \{a, b\}$, $N = \{S, A, B\}$ and $P$ is the set*

$$\{A \mapsto a, A \mapsto AA, B \mapsto b, B \mapsto BB, S \mapsto AB\}$$

*$X = [x_1, x_2, x_3]$. Both tuples $(a, a, b)$ and $(a, b, b)$ satisfies the constraint* $\texttt{grammar}(X, G)$ *since the corresponding sequences are in the grammar $G$.*

Hard global constraints can be reformulated to soft global constraints, by associating violation measures. This technique is called *constraint softening* [45].

**Definition 2.20.** (Constraint softening) *Let $C_S^h$ be a hard constraint and $\mu$ is a* violation measure *of $C_S^h$ which maps a tuple $l \in \mathcal{L}(S)$ to a cost. Then* $\texttt{soft\_}C_S^\mu$ *is defined as a soft constraint that for all tuples $l \in \mathcal{L}(S)$:*

$$\texttt{soft\_}C_S^\mu(l) = \begin{cases} 0, & l \text{ satisfies } C_S^h \\ \mu(l), & \text{otherwise} \end{cases}$$

Note that a soft global constraint can be associated with more than one violation measures. In the context where the violation measure $\mu$ of the soft constraint is not important, we denote the constraint simply by $C_S$. Common violation measures including the following.

**Definition 2.21.** *Let $C_S^h$ be a hard constraint and $l$ is a tuple in $\mathcal{L}(S)$.*

- *[41] The* variable-based violation measure *maps $l$ to the minimum number of variable assignments required to change in $l$ to satisfy $C_S^h$.*

- *[49] The* edit-based violation measure *maps $l$ to the minimum number of insertions, deletions, and substitutions required to change $l$ into a tuple satisfies $C_S^h$.*

- *[41]Assume $C_S^h$ can be decomposed into a set of constraints $C_{dec}$. The* decomposition-based violation measure *of $C_S^h$ maps $l$ to the number of constraints in $C_{dec}$ violates by $l$.*

For convenience we also define the *constant violation measure* as a function maps a tuple to a constant cost.

**Example 2.16.** *Let $C_S^h$ be the hard constraint* $\texttt{among}(S, 1, 2, \{a, b\})$. *The soft version of $C_S^h$ with variable-based violation measure is* $\texttt{soft\_among}^{var}(S, 2, 3, \{a\})$. *The soft constraint on the tuple $(b, a, c)$ returns a cost 1 since either the first or the last component of the tuple has to be changed to a in order to satisfy $C_S^h$.*

Throughout this thesis we always assume the representation size of a global constraint is polynomial in the number of variables restricted by the constraint and the maximum size of the variable domains.

# Chapter 3

# Tractable Projection-Safety

We say a soft constraint is *tractable* if the computation of its minimum cost can be done in time polynomial in the representation size of the global constraint (i.e. in the number of variables and maximum domain size). Tractability of a soft constraint is important. As we have seen many examples of WCSP consistencies in the last chapter, those consistencies cannot be enforced in polynomial time unless the constraints in the system are tractable. However, consistency algorithms also modify constraints by projections and extensions. Even if a constraint is tractable, it is not guaranteed that the resultant constraint obtained after projections and extensions is tractable. In this chapter we address this issue, namely *tractable projection-safety*. Our discussion is divided into three cases of projections and extensions for constraints of different arities. The result shows that projections and extensions indeed hinders the tractability of a constraint in some cases.

In the second part of this chapter we will discuss a class of constraints, namely *polynomially decomposable* constraints. Our technique sequentially decomposes a constraint into smaller and smaller constraints. The number of constraints appear in the sequence is bounded, and each constraint is tractable. We can compute the minimum cost of the original constraint from the minimum costs of these smaller constraints, and maintain that the decomposition still holds after projections and extensions. We show that a polynomially

decomposable constraint is tractable projection-safe. Given a polynomially decomposable constraint. We show that we can apply a dynamic programming algorithm to compute its minimum cost, and the algorithm still works after projections and extensions.

## 3.1 Tractable Projection-Safety: Definition and Analysis

In the following, we denote the minimum cost of a constraint $C_S$, $\min\{C_S(l)|l \in \mathcal{L}(S)\}$, by $\min(C_S)$. Note that enforcement algorithms of WCSP consistencies usually query a part of a constraint. For example (G)AC* enforcement algorithm queries the minimum cost when a variable is fixed to a value, say $\min\{C_S(l)|l \in \mathcal{L}(S) \wedge l[x_i] = v\}$ for a variable $x_i \in S$ and a value $v \in D(x_i)$. As long as there is an efficient algorithm to compute $\min(C_S)$, this value can be computed by simply assuming $D(x_i) = \{v\}$ and then computing $\min(C_S)$.

Following Leung [32], the general notion of projection-safety is defined as follows. Let $\mathcal{T}$ be an arbitrary property and $r$ a non-negative integer. A soft constraint $C_S$ is $\mathcal{T}$ $r$-projection-safe if:

- $C_S$ satisfies the property $\mathcal{T}$, and;
- $\Delta_r(C_S)$ satisfies the property $\mathcal{T}$, for all series of $r$-projections/$r$-extensions $\Delta_r$.

A soft constraint $C_S$ is $\mathcal{T}$ $r$-projection-safe means that the constraint preserves the property $\mathcal{T}$ even after projections and extensions. The property we concern about is tractability. A soft constraint $C_S$ is *tractable* if there exists an algorithm to compute its minimum cost $\min(C_S)$, and runs in polynomial time.

A soft constraint $C_S$ is *tractable $r$-projection-safe* if

- $C_S$ is tractable, and;
- $\Delta(C_S)$ is tractable, where $\Delta_r$ is a series of $r$-projections/$r$-extensions $\Delta_r$.

Remind that a table constraint is a constraint represented as a table, where each entry of the table corresponding to a possible assignment, specifying its cost. The following theorem shows that a table constraint is always tractable.

**Theorem 3.1.** *Table constraints are tractable and tractable $r$-projection-safe for an fixed $r$.*

*Proof.* A table constraint is tractable because we can scan the minimum cost of every possible assignments on the table and return the minimum one. This take time linear in the table size.

The constraint obtained from applying a series of $r$-projections and $r$-extensions $\Delta_r$ can be represented as a table constraint. Then the same algorithm to compute minimum cost applies. □

We mainly focus on soft global constraints. In the following we divide the discussion of tractable $r$-projection-safety into three cases of different $r$: (a) $r = 0$, (b) $r \geq 2$ and (c) $r = 1$.

**Case 1:** $r = 0$. In this case, projections and extensions are only to/from $C_\varnothing$.

**Theorem 3.2.** *A tractable constraint $C_S$ is tractable 0-projection-safe.*

*Proof.* Let $C_S$ $\Delta_0$ a series of 0-projections/0-extensions from/to $C_S$. Note that if $l_{\min}$ is the minimum cost tuple in $C_S$, i.e. $\min(C_S) = C_S(l_{\min})$, $l_{\min}$ is also a minimum cost tuple in $\Delta_0(C_S)$. We can first compute $\min(C_S)$ then evaluate $\min(\Delta_0(C_S)) = \Delta_0(C_S)(l_{\min})$. □

0-projections/0-extensions are employed in ∅IC [51] and strong ∅IC [30] enforcements. Consequently, as long as all the constraints in the system are tractable, enforcement algorithms for (strong) ∅IC runs in polynomial time.

Although (strong) ∅IC are relatively weaks form of WCSP consistency, they can be efficiently applied to a wide range of soft constraints.

**Case 2:** $r \geq 2$. In this case, projections and extensions are to/from $r$-arity constraints. The key observation here is that, by using projections/extensions of arity $r$, we can encode any relations (constraints) between variables into the newly obtained constraint.

**Theorem 3.3.** *A tractable soft global constraint $C_S$ is not tractable $r$-projection-safe for $2 \leq r \leq |S|$.*

*Proof.* We use a reduction from CSP. Given a CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}^h)$ where every constraint in $\mathcal{C}^h$ are $r$-arity. We construct $C_S = C_{\mathcal{X}}$. Let $k$ be the maximum cost of an assignment in $C_S$, i.e. $k = \max(C_S)$. We construct a series of $r$-extension $\Delta_r$ as follows. Initially $\Delta_r$ is empty. For each hard constraint $C^h_S$, and any tuple $l$ unsatisfying $C^h_S$, we append an extension of cost $k+1$ to $C_{\mathcal{X}}$. $\mathcal{P}$ is satisfiable if and only if $\min(\Delta_r(C_{\mathcal{X}})) \leq k$, because any tuple $l$ unsatisfying a constraint $C^h_S \in \mathcal{C}$ must incur a cost of at least $k+1$ in $C_{\mathcal{X}}$. □

Theorem 3.3 shows that in general, even if a constraint $C_S$ is tractable, $C_S$ is not tractable after projections to or extensions from $r$-arity constraints. Projections and extensions of arity larger than 1 are required for enforcing consistencies in ternary constraints [47] and $k$-consistency [16]. Thus, these consistency techniques are hard to apply efficiently to global constraints.

**Case 3:** $r = 1$. A soft constraint $C_S$ is *flow-based* [49] if it can be represented by a flow network $G$ such that the minimum cost flow on $G$ corresponds to the minimum cost of $C_S$. A soft constraint $C_S$ is flow-based projection-safe if:

- $C_S$ is flow-based, and;
- $\Delta_1(C_S)$ is flow-based for all series of 1-projections/1-extensions $\Delta_1$.

Theorem 3.4 gives a sufficient condition for flow-based projection-safety, which is a special case of 1-projection safety. By using this theorem, it is shown that $\texttt{soft\_allDifferent}^{dec}$, $\texttt{soft\_allDifferent}^{var}$, $\texttt{soft\_gcc}^{var}$, $\texttt{soft\_gcc}^{val}$, $\texttt{soft\_same}^{var}$, $\texttt{soft\_regular}^{var}$ and $\texttt{soft\_regular}^{edit}$ are all flow-based projection-safe [30, 32].

**Theorem 3.4.** *[30, 32] Given a soft global constraint $C_S$ such that:*

- *$C_S$ is flow-based, with the corresponding network $G$;*
- *there exists a function $\Phi$ mapping each maximum flow $f$ in $G$ to each tuple $\Phi(f) \in \mathcal{L}$, and;*
- *there exists an injection from an assignment $\{x_i \mapsto v\}$ to a subset of edges $\bar{E}$ of the edge set of $G$, such that whenever $l[x_i] = v$ for some tuple $l$, $\sum_{e \in \bar{E}} f_e = 1$ in the flow corresponding to $l$; whenever $l[x_i] \neq v$, $\sum_{e \in \bar{E}} f_e = 0$.*

*$C_S$ is flow-based projection-safe.*

**Theorem 3.5.** *A flow-based projection-safe constraint $C_S$ is tractable* 1*-projection-safe.*

*Proof.* Let $\Delta_1$ a series of 1-projections/1-extensions. By definition $\Delta_1(C_S)$ is flow-based, and by finding a minimum cost network flow on the corresponding flow network $G$ we can compute $\min(\Delta_1(C_S))$ in polynomial time. □

We also observe that tractable constraints are not necessarily tractable 1-projection-safe. The $\texttt{soft\_2sat}^{const}$ constraint is an example. Given a set of boolean variable $X$, a set of binary clauses $F$ and a constant $c \in [0 \dots k]$. The $\texttt{soft\_2sat}^{const}(X, F, c)$ constraint is a soft constraint defined as:

$$\texttt{soft\_2sat}^{const}(X, F, c)(l) = \begin{cases} 0, & \text{if } l \text{ satisfies } F \\ c, & \text{if } l \text{ does not satisfy } F \end{cases}$$

Computing $\min(\texttt{soft\_2sat}^{const}(X, F, c))$ is equivalent to determining the 2-boolean satisfiability, which is tractable [25]. Thus $\texttt{soft\_2sat}^{const}$ is tractable. However, $\texttt{soft\_2sat}^{const}$ is not tractable 1-projection-safe as explained below.

Let $F$ be a set of binary clauses, $X$ the set of boolean variables of $F$, and $k$ a non-negative integer. The problem W2SAT is to determine whether these exists a assignment to $X$ satisfying $F$, with at most $k$ variables set to *true*. W2SAT is NP-Hard [20]. We use a reduction from W2SAT to computing the minimum cost of a constraint obtained from applying a series of 1-projections/1-extensions to $\texttt{soft\_2sat}^{const}$.

**Theorem 3.6.** *The* $\texttt{soft\_2sat}^{const}$ *constraint is not tractable* 1*-projection-safe, unless* $P = NP$.

*Proof.* Given a set of binary clauses $F$, a set of boolean variables $X$, and a non-negative integer $k$. We construct an $\texttt{soft\_2sat}^{const}$ constraint $C_S$ and a series of 1-projections/1-extensions $\Delta_1$ as follows. $C_S$ is the constraint $\texttt{soft\_2sat}^{const}(X, F, k+1)$. $\Delta_1$ is initially empty. For each variable $x_i \in X$, we add a 1-projection of cost 1 from $C_i(true)$ to $C_S$, where $C_i$ is a unary constraint over $x_i$. Every assignment to $X$ satisfying $F$ incurs a cost of $k'$ in $\Delta_1(C_S)$ where $k'$ is the number of variables assigned to *true* in the assignment. Thus, by determining whether $\min(\Delta_1(C_S)) \leq k$ we can solve the W2SAT problem. □

According to the above discussion, tractable constraints are tractable 1-projection-safe only under special conditions. 1-projections/1-extensions are the backbone of the consistency algorithms of (G)AC* [18, 30], FD(G)AC*[27, 30] and (weak) ED(G)AC* [19, 31]. Thus, these consistency techniques can be efficiently apply to 1-projection-safe global constraints.

To summarize, given a tractable soft constraint $C_S$, $C_S$ must be tractable 0-projection-safe, $C_S$ cannot be tractable $r$-projection-safe with $r \geq 2$, and $C_S$ *may be* tractable 1-projection-safe. To simplify notations, we write *tractable*

*projection-safe* to mean tractable 1-projection-safe in the rest of the paper. Lee and Leung [30] gives only one sufficient condition for tractable projection-safety based on flow-based global constraints. In the next section, we will show another sufficient condition based on another type of tractable constraints, namely *polynomially decomposable* constraints.

## 3.2 Polynomially Decomposable Soft Constraints

In this section we introduce a new class of tractable projection-safe constraints, namely *polynomially decomposable constraints*, which are derived from abstracting dynamic programming algorithms that compute the minimum cost of a constraint. These algorithms often imply decompositions of the constraints. Examples of polynomially decomposable constraints are given in Chapter 4. For convenience, we write projections/extensions to mean 1-projections/1-extensions in this section.

A constraint $C_S$ *safely decomposes* into a sequence of constraints $[C_{S_1}, C_{S_2}, \ldots, C_{S_m}]$ where $S_i \subseteq S$ and $f$ is a polynomial time computable function, such that:

- $C_S(l) = f(C_{S_1}(l[S_1]), \ldots, C_{S_m}(l[S_m]))$ holds for all assignments $l$, and;
- for any constraint $C'_S$ and sequence of constraints $[C'_{S_1}, \ldots, C'_{S_m}]$, where $C'_S(l) = f(C'_{S_1}(l[S_1]), \ldots, C'_{S_m}(l[S_m]))$, it holds that:
  - (a) $\min(C'_S) = f(\min(C'_{S_1}), \ldots, \min(C'_{S_m}))$, and;
  - (b) for a variable $x \in S$, a cost $\alpha$ and a complete assignment $l^*$, we have:

$$C'_S(l^*) \oplus \alpha = f(C'_{S_1}(l^*[S_1]) \oplus \nu_{x,S_1}(\alpha), \ldots, C'_{S_m}(l^*[S_m]) \oplus \nu_{x,S_m}(\alpha))$$
$$C'_S(l^*) \ominus \alpha = f(C'_{S_1}(l^*[S_1]) \ominus \nu_{x,S_1}(\alpha), \ldots, C'_{S_m}(l^*[S_m]) \ominus \nu_{x,S_m}(\alpha))$$

The function $\nu$ is defined as:

$$\nu_{x,S}(\alpha) = \begin{cases} \alpha, & x \in S \\ 0, & \text{otherwise} \end{cases}$$

In another word, $C_S$ can be represented as a combination of $C_{S_1}, \ldots, C_{S_m}$. In addition, condition (a) allows us to compute $\min(C_S)$ from $\min(C_{S_1}), \ldots, \min(C_{S_m})$. Condition (b) suggests how projections and extensions on $C_S$ can be distributed to its components.

Let $\delta$ be a projection of cost $\alpha$ from $C_i(v)$, or extension from $C_i(v)$ where $C_i$ is a unary constraint over a variable $x_i \in S$. We have the following result. We only proof the part of extension, while the proof on projection is similar.

**Theorem 3.7.** *Let $C_S$ be a constraint safely decomposes into $[C_{S_1}, \ldots, C_{S_m}]$. $\delta(C_S)$ safely decomposes into $[\delta(C_{S_1}), \ldots, \delta(C_{S_m})]$.*

*Proof.* It is sufficient to prove $\delta(C_S)(l) = f(\delta(C_{S_1})(l[S_1]), \ldots, \delta(C_{S_m})(l[S_m]))$ holds for all assignments $l$. In case $l[x_i] \neq v$, $\delta(C_S)(l) = C_S(l)$ and $\delta(C_{S_i})(l) = C_{S_i}(l[S_i])$ for all $1 \leq i \leq m$ and result follows. Suppose $l[x_i] = v$.

$$\begin{aligned} \delta(C_S)(l) &= C_S(l) \oplus \alpha \\ &= f(C_{S_1}(l[S_1]) \oplus \nu_{x_i,S_1}(\alpha), \ldots, C_{S_m}(l[S_m]) \oplus \nu_{x_i,S_m}(\alpha)) \\ &= f(\delta(C_{S_1})(l[S_1]), \ldots, \delta(C_{S_m})(l[S_m])) \end{aligned}$$

□

Theorem 3.8 and Theorem 3.9 give special scenarios where a constraint can be safely decomposed.

**Theorem 3.8.** *Given a constraint $C_S$ and a sequence of constraints $[C_{S_1} \ldots C_{S_m}]$, where for all $1 \leq i \leq m$, $S_i \subseteq S$, and for all $j \neq k, S_j \cap S_k = \varnothing$, satisfying*

$$C_S(l) = \bigoplus_{1 \leq i \leq m} C_{S_i}(l[S_i])$$

*for all possible assignments $l$. $C_S$ safely decomposes into $C_{S_1}, \ldots, C_{S_m}$.*

*Proof.* Let $l_{\min}$ be an assignment on $S$ such that $C_S(l_{\min}) = \min(C_S)$, and $l_i$ an assignment on $S_i$ such that $C_{S_i}(l_i) = \min(C_{S_i})$ for $1 \leq i \leq m$. We claim $C_{S_u}(l_{\min}[S_u]) = C_{S_u}(l_u)$ for $1 \leq u \leq m$. Otherwise, $C_S(l_{\min}[S \setminus S_u] \cup l_u) = C_{S_u}(l_u) \oplus \bigoplus_{1 \leq i \leq m, i \neq u} C_{S_i}(l_i) < C_{S_u}(l[S_u]) \oplus \bigoplus_{1 \leq i \leq m, i \neq u} C_{S_i}(l_i) = C_S(l_{\min})$ which leads to contradiction. So we have

$$\min(C_S) = \bigoplus_{1 \leq i \leq m} C_{S_i}(l_i) = \bigoplus_{1 \leq i \leq m} \min(C_{S_i})$$

On the other hand, given a variable $x \in S$, a cost $\alpha$, and a assignment $l$ on $S$. $x$ is in the scope of at most one constraint. Suppose the constraint is $C_{S_v}$. We have

$$C_S(l) \oplus \alpha = (C_{S_v}(l[S_v]) \oplus \alpha) \oplus \bigoplus_{1 \leq i \leq m, i \neq v} C_{S_i}(l[S_i])$$

For $\ominus$ it is similar. □

**Theorem 3.9.** *Given a constraint $C_S$ and a sequence of constraints $[C_{S_1}, \ldots, C_{S_m}]$, where for all $1 \leq i \leq m$, $S_i = S$, satisfying*

$$C_S(l) = \min_{1 \leq i \leq m} C_{S_i}(l)$$

*for all possible assignments $l$. $C_S$ safely decomposes into $C_{S_1}, \ldots, C_{S_m}$.*

*Proof.* Let $\mathcal{L}(S)$ be the set of all possible assignments to variables in $S$. We have

$$\begin{aligned} \min(C_S) &= \min_{l \in \mathcal{L}(S)} \{\min_{1 \leq i \leq m} C_{S_i}(l)\} \\ &= \min_{1 \leq i \leq m} \{\min_{l \in \mathcal{L}(S)} C_{S_i}(l)\} \\ &= \min_{1 \leq i \leq m} \{\min(C_{S_i})\} \end{aligned}$$

On the other hand, given a variable $x \in S$, a cost $\alpha$, and a assignment $l$ on $S$. $\nu_{x,S_i}(\alpha) = \alpha$ holds for all $1 \leq i \leq m$. We have

$$\begin{aligned} C_S(l) \oplus \alpha &= \min_{1 \leq i \leq m} \{C_{S_i}(l) \oplus \alpha\} \\ &= \min_{1 \leq i \leq m} \{C_{S_i}(l) \oplus \alpha\} \end{aligned}$$

For $\ominus$ it is similar. □

**Theorem 3.10.** *Given a constraint $C_S$ satisfying*

$$C_S(l) = \min_{1 \le i \le m} \{ \bigoplus_{1 \le j \le n_i} C_{S_{i,j}}(l[S_{i,j}]) \}$$

*for all possible assignments $l \in \mathcal{L}(S)$, where $m$ and $n_i$ for $1 \le i \le m$ are positive integers, $C_{S_{i,j}}$ are constraints, and $S = \bigcup_{1 \le j \le n_i} S_{i,j}$ for $1 \le i \le m$. $C_S$ safely decomposes into $C_{S_{1,1}}, \ldots, C_{S_{m,n_m}}$.*

*Proof.* We prove the result by creating redundant constraints $C_S^i$ satisfying

$$C_S^i(l) = \bigoplus_{1 \le j \le n_i} C_{S_{i,j}}(l[S_{i,j}])$$

for all $l \in \mathcal{L}(S)$ where $1 \le i \le m$. By Theorem 3.8, $C_S^i$ safely decomposes into $C_{S_{i,1}}, \ldots, C_{S_{i,n_i}}$. By Theorem 3.9 $C_S$ safely decomposes into $C_S^1, \ldots, C_S^m$. Thus, we have

$$\begin{aligned} \min(C_S) &= \min_{1 \le i \le m}\{\min(C_S^i)\} \\ &= \min_{1 \le i \le m}\{\bigoplus_{1 \le j \le n_i}\{\min(C_{S_{i,j}})\}\} \end{aligned}$$

Also, with a variable $x \in S$, a cost $\alpha$, and a assignment $l$ on $S$, we have

$$\begin{aligned} C_S(l) \oplus \alpha &= \min_{1 \le i \le m}\{C_S^i(l) \oplus \alpha\} \\ &= \min_{1 \le i \le m}\{\bigoplus_{1 \le j \le n_i}\{C_{S_{i,j}}(l[S_{i,j}]) \oplus \nu_{x,S_{i,j}}(\alpha)\}\} \end{aligned}$$

For $\ominus$ it is similar. □

A constraint that can be safely decomposed is not necessary tractable. For one thing, safe decomposition does not required that each constraint in the sequence is tractable. For another, the length of the sequence is not bounded.

Given a constraint $C_S$ on the set of variables $S$. The constraint $C_S$ is *polynomially decomposable* if there is a sequence of constraints $[C_{S_1}, C_{S_2}, \ldots, C_{S_m}]$ such that:

- $C_S = C_{S_m}$ and $m$ is polynomial in $|S|$ and the maximum size of the variable domains in $S$, and;

- Each $C_{S_i}$ is either a tractable unary constraint, or can be safely decomposed into $[C'_{i,1}, C'_{i,2}, \ldots, C'_{i,m_i}]$ where $m_i < i$ and $C'_{i,j} \in \{C_{S_1}, \ldots, C_{S_{i-1}}\}$ for all $j \leq m_i$.

Given a series of 1-projections/1-extensions $\Delta$, and a polynomially decomposable constraint $C_S$. In the following, we show that $C_S$ is tractable, and so is $\Delta(C_S)$. Thus, $C_S$ is tractable projection-safe.

**Lemma 3.11.** *A polynomially decomposable constraint $C_S$ is tractable.*

*Proof.* Let $[C_{S_1}, \ldots, C_{S_m}]$ be a sequence of constraints where $C_S = C_{S_m}$ and $m$ is polynomial in $|S|$ and the maximum size of the variable domains in $S$, and each $C_{S_i}$ is either a tractable unary constraint and can be safely decomposed into $[C'_{i,1}, \ldots, C'_{i,m_i}]$ where $m_i < i$ and $C'_{i,j} \in \{C_{S_1}, \ldots, C_{S_{i-1}}\}$ for all $j \leq m_i$. By definition such a sequence exists. Algorithm 3.1 can be applied to compute the minimum cost of a polynomially decomposable constraint. The algorithm uses a dynamic programming approach, loops through the sequence and computes the minimum cost of each constraint appears. An associative array `MinCost` is used to store minimum costs of each constraint in the decomposed sequence to avoid re-computation. It remains to analyze the run time. Each $\min(C_{S_i})$ is evaluated at most once in polynomial time. Since the sequence is polynomial in size, result follows. □

**Algorithm 3.1**: Compute minimum cost of $C_S$

```
1 for i ← 1 to m do
2     MinCost[C_{S_i}] ← f(MinCost[C'_{i,1}], ..., MinCost[C'_{i,m_i}]) ;
3 return MinCost[C_S] ;
```

**Theorem 3.12.** *A polynomially decomposable constraint $C_S$ is tractable projection-safe.*

*Proof.* Let $[C_{S_1}, \ldots, C_{S_m}]$ be the corresponding sequence and each $C_{S_i}$ safely decomposes into $[C'_{i,1}, \ldots, C'_{i,m_i}]$. $\delta$ is a 1-projection/1-extension. $\delta(C_S)$ is polynomially decomposable since by Theorem 3.7, $\delta(C_{S_i})$ safely decomposes into $[\delta(C'_{i,1}), \ldots, \delta(C'_{i,m_i})]$. By induction after a series of projections/extensions $\Delta$, $\Delta(C_S)$ is polynomially decomposable. By Lemma 3.11, $\Delta(C_S)$ is also tractable. Result follows.

□

Theorem 3.12 gives rise to a class of constraints that is tractable projection-safe. Algorithm 3.1 is the basis of an efficient dynamic programming algorithm to compute the minimum cost of a polynomially decomposed constraint.

In the next chapter, we will give several examples of polynomially decomposable constraints.

# Chapter 4

# Examples of Polynomially Decomposable Soft Global Constraints

As we have shown in the last chapter, polynomially decomposable constraints are tractable projection-safe. In this chapter, we give examples of polynomially decomposable constraints, including soft variant of `among`, `regular`, `grammar` constraints, and `max_weight` constraint. Thus, these constraints are tractable projection-safe. Depending on the decomposition, we give an algorithm for each of these constraints to compute the minimum cost. Moreover, after a series of 1-projections/1-extensions we can still use the same algorithm to compute the minimum cost. Thus, the algorithms allow us to efficiently enforce consistencies which depends on 1-projections/1-extensions. Note that these algorithms are special cases of Algorithm 3.1.

In the following, we use $n$ to denote the number of variables involved in a constraint $C_S$, and $d$ the maximum domain size, i.e. $d = \max_{x_i \in S}\{|D(x_i)|\}$. Also, we assume the variables in $S = \{x_1, \ldots, x_n\}$ are ordered by their indices.

## 4.1 Soft Among Constraint

Given a set of variables $S$, a set of values $V$, a lower bound $lb$ and a upper bound $ub$, where $lb \leq ub$. We define $t(l) = |\{i|l[x_i] \in V\}|$ as the number of variables taking a value in $V$ in the assignment $l$. The $\texttt{soft\_among}^{var}(S, lb, ub, V)$ constraint [7] with the variable-based violation measure is a constraint defined as:

$$\texttt{soft\_among}^{var}(S, lb, ub, V)(l) = \max(0, lb - t(l), t(l) - ub)$$

**Theorem 4.1.** *The* $\texttt{soft\_among}^{var}$ *constraint is polynomially decomposable.*

*Proof.* Let $C_S$ be the constraint $\texttt{soft\_among}^{var}(S, lb, ub, V)$ where $S = \{x_1 \dots x_n\}$. Let $S_i = \{x_1, \dots, x_i\}$ and in particular $S_0 = \varnothing$. We denote the constraint $\texttt{soft\_among}^{var}(S_i, j, j, V)$ by $C^j_{S_i}$. In particular $C^j_{S_0}$ always returns $j$ by definition. We also define the unary constraint $U^k_i$ on variable $x_i \in S$ where $k \in \{0, 1\}$ that for all $v \in D(x_i)$:

$$U^k_i(v) = \begin{cases} 0, & \text{if } (k = 0 \wedge v \notin V) \vee (k = 1 \wedge v \in V) \\ 1, & \text{otherwise} \end{cases}$$

We show $C_S$ to be polynomially decomposable by constructing a sequence $[U^k_i, \dots, C^j_{S_i}, \dots, C_S]$. In the sequence $\{C^j_{S_i}\}$ are ordered in the increasing order of $i$. The length of the sequence is bounded by $O(nd)$. Considering $C^j_{S_i}$ where $i > 0$, if the last variable in its scope $x_i$ takes a value in (not in) $V$, it requires the variables in the set $S_{i-1}$ has $j - 1$ ($j$) values in $V$. Thus, for all assignments $l$:

$$C^0_{S_i}(l[S_i]) = C^0_{S_{i-1}}(l[S_{i-1}]) \oplus U^0_i(l[x_i])$$
$$C^j_{S_i}(l[S_i]) = \min \begin{cases} C^{j-1}_{S_{i-1}}(l[S_{i-1}]) \oplus U^1_i(l[x_i]) \\ C^j_{S_{i-1}}(l[S_{i-1}]) \oplus U^0_i(l[x_i])) \end{cases}$$

for $i > 0$ and $j > 0$. Finally, by definition, for all assignments $l$:

$$C_S(l) = \min_{lb \leq j \leq ub} C^j_{S_n}(l)$$

By Theorem 3.10, each constraint in the constructed sequence is either unary constraint, or can be safely decomposes into constraints precede it in the sequence. □

**Example 4.1.** *Suppose* $C_S$ *is the constraint* $\texttt{soft\_among}^{var}(S, 0, 1, \{a\})$ *where* $S = \{x_1, x_2, x_3\}$. $C^j_{S_i}$ *and* $U^k_i$ *are defined as above. We construct the sequence*

$$[U^0_1, U^1_1, U^0_2, U^1_2, U^0_3, U^1_3, C^0_{S_1}, C^1_{S_1}, C^0_{S_2}, C^1_{S_2}, C^0_{S_3}, C^1_{S_3}, C_S]$$

*For all assignments* $l \in \mathcal{L}(S)$*, we have:*

$$\begin{aligned}
C^0_{S_1}(l[S_1]) &= U^0_i(l[x_1]) \\
C^1_{S_1}(l[S_1]) &= \min(U^1_i(l[x_1]), 1 \oplus U^0_i(l[x_1])) \\
C^0_{S_2}(l[S_2]) &= C^0_{S_1}(l[S_1]) \oplus U^0_2(l[x_2]) \\
C^1_{S_2}(l[S_2]) &= \min(C^0_{S_1}(l[S_1]) \oplus U^1_2(l[x_2]), C^1_{S_1}(l[S_1]) \oplus U^0_2(l[x_2])) \\
C^0_{S_3}(l) &= C^0_{S_2}(l[S_2]) \oplus U^0_3(l[x_3]) \\
C^1_{S_3}(l) &= \min(C^0_{S_2}(l[S_2]) \oplus U^1_3(l[x_3]), C^1_{S_2}(l[S_2]) \oplus U^0_2(l[x_3])) \\
C_S(l) &= \min(C^0_{S_3}(l), C^1_{S_3}(l))
\end{aligned}$$

**Corollary 4.2.** *The* $\texttt{soft\_among}^{var}$ *constraint is tractable projection-safe.*

*Proof.* Result follows from Theorem 3.12 and Theorem 4.1. □

**Algorithm 4.1**: Computing the minimum cost of $\texttt{soft\_among}^{var}$

**Input**: $C_S$ : $\texttt{soft\_among}^{var}(S, lb, ub, V)$, and a series of 1-projections/1-extensions $\Delta$
**Output**: $\min(\Delta(C_S))$

```
1 for i ← 1 to n do
2     for j ∈ {0,1} do u_i^j ← min(Δ(U_i^j)) ;
3 for j ← 0 to ub do
4     f_0^j ← j ;
5 for i ← 1 to n do
6     f_i^0 ← f_{i-1}^0 ⊕ u_i^0 ;
7     for j ← 1 to ub do
8         f_i^j ← min(f_{i-1}^{j-1} ⊕ u_i^1, f_{i-1}^j ⊕ u_i^0) ;
9 return min_{lb≤j≤ub}{f_n^j} ;
```

Algorithm 4.1 computes the minimum cost of a $\texttt{soft\_among}^{var}$ constraint. In the algorithm, it holds that $u_i^j = \min(U_i^j)$ and $f_i^j = \min(C_i^j)$. Line 1-2 computes the minimum cost of each unary constraint $U_i^j$. As the unary constraints are stored as tables, the minimum costs can be computed by scanning the table. This step is done in $O(nd)$ time, since there are $2n$ such unary constraints and the domain size of each variable is at most $d$. Line 3-4 initialize $f_0^j = j$ for $0 \leq j \leq ub$, since $C_0^j$ always returns $j$. The algorithm then computes the minimum costs of rest of the constraints in the sequence. By definition of safe decomposition, and the decomposition shown in the proof of Theorem 4.1, we have

$$\min(\Delta(C_i^0)) = \min(\Delta(C_{i-1}^0)) \oplus \min(\Delta(U_i^0))$$
$$\min(\Delta(C_i^j)) = \min\{\min(\Delta(C_{i-1}^{j-1}) \oplus \min(\Delta(U_i^1)), \min(\Delta(C_{i-1}^j) \oplus \min(\Delta(U_i^0))\}$$

Line 5-8 of the algorithm computes the minimum costs accordingly, in $O(n^2)$ time. Finally, the algorithm returns the answer base on the fact that

$$\min(\Delta(C_S)) = \min_{lb \leq j \leq ub}\{\min(\Delta(C_n^j))\}$$

This final step takes time $O(n)$.

**Theorem 4.3.** *Given a* $\texttt{soft\_among}^{var}$ *constraint and a series of* 1*-projections/*1*-extensions* $\Delta$*. Algorithm 4.1 computes the minimum cost of* $\texttt{soft\_among}^{var}$ *constraint after applying* $\Delta$*, in* $O(n^2 + nd)$ *time.*

## 4.2 Soft Regular Constraint

$\texttt{soft\_regular}^{var}$ is the soft form of the `regular` constraint with the variable-based violation measure. In addition, when it is impossible to satisfy the underlying `regular` constraint, the cost is $\top$. Leung [32] has shown that the $\texttt{soft\_regular}^{var}$ constraint is flow-based projection-safe and thus tractable projection-safe. We are to show that this constraint is also polynomially decomposable and derive another algorithm to compute its minimum cost.

**Theorem 4.4.** *The* $\texttt{soft\_regular}^{var}$ *constraint is polynomially decomposable.*

*Proof.* Let $C_S$ be the constraint $\texttt{soft\_regular}^{var}(S, M)$ where $S = \{x_1, \ldots, x_n\}$ and $M = (Q, \Sigma, T, q_0, F)$ (see Definition 2.18). Let $S_i = \{x_1, \ldots, x_i\}$. For each state $q_j \in Q$, we define a deterministic finite automaton $M_j = (Q, \Sigma, T, q_0, \{q_j\})$, with $q_j$ being the only final state. We denote a constraint $\texttt{soft\_regular}^{var}(S_i, M_j)$ by $C^j_{S_i}$. In particular, $C^0_{S_0}$ always returns 0 and $C^j_{S_0}$ always returns $\top$ for $j \neq 0$. We also define the unary constraint $U^c_i$ on the variable $x_i$ where $c \in \Sigma$ that for all $v \in D(x_i)$:

$$U^c_i = \begin{cases} 0, & \text{if } v = c \\ 1, & \text{otherwise} \end{cases}$$

We show $C_S$ to be polynomially decomposable by constructing a sequence $[U^c_i, \ldots, C^j_{S_i}, \ldots, C_S]$. In the sequence $\{C^j_{S_i}\}$ are ordered in the increasing order of $i$. The length of the sequence is bounded by $O(n \cdot |M|)$. Consider $C^j_{S_i}$ where $i > 0$. For each transition $(q_k, c) \mapsto q_j$, in order to form a sequence recognizable by $M_j$, we can choose to make the variables in $S_{i-1}$ forms a sequence recognizable by $M_k$, and the variable $x_i$ to be $c$. Thus, for all assignments $l \in \mathcal{L}(S)$:

$$C^j_{S_i}(l[S_i]) = \min_{(q_k,c)\mapsto q_j \in T} \{C^k_{S_{i-1}}(l[S_{i-1}]) \oplus U^c_i(l[x_i])\}$$

for $i > 0$ and $q_j \in Q$. Finally, the set of sequences recognizable by $M$ are the union of the set of sequences recognizable by $M_j$ for all $q_j \in F$. Thus, for all assignments $l$:

$$C_S(l) = \min_{q_j \in F} C^j_{S_n}(l)$$

By Theorem 3.10, each constraint in the constructed sequence is either unary constraint, or can be safely decomposes into constraints precede it in the sequence. □

**Example 4.2.** *Suppose* $C_S$ *is the constraint* $\texttt{soft\_regular}^{var}(S, M)$ *where* $S = \{x_1, x_2, x_3\}$ *and* $M$ *is the finite state automaton shown in Figure 2.12.*

*$C^j_{S_i}$ and $U^c_i$ are defined as above. We construct the sequence*

$$\begin{aligned}&[U^a_1, U^b_1, U^a_2, U^b_2, U^a_3, U^b_3,\\ &C^0_{S_1}, C^1_{S_1}, C^2_{S_1}, C^3_{S_1},\\ &C^0_{S_2}, C^1_{S_2}, C^2_{S_2}, C^3_{S_2},\\ &C^0_{S_3}, C^1_{S_3}, C^2_{S_3}, C^3_{S_3}, C_S]\end{aligned}$$

*For all assignments $l \in \mathcal{L}(S)$ we have:*

$$\begin{aligned}
C^0_{S_1}(l[S_1]) &= \top\\
C^1_{S_1}(l[S_1]) &= \min\{\top, C^0_{S_0} \oplus U^a_1(l[x_1]), C^1_{S_0} \oplus U^a_1(l[x_1])\}\\
C^2_{S_1}(l[S_1]) &= \min\{\top, C^1_{S_0} \oplus U^b_1(l[x_1])\}\\
C^3_{S_1}(l[S_1]) &= \min\{\top, C^0_{S_0} \oplus U^b_1(l[x_1]), C^3_{S_0} \oplus U^a_1(l[x_1])\}\\
C^0_{S_2}(l[S_2]) &= \top\\
C^1_{S_2}(l[S_2]) &= \min\{\top, C^0_{S_1}(l[S_1]) \oplus U^a_2(l[x_2]), C^1_{S_1}(l[S_1]) \oplus U^a_2(l[x_2])\}\\
C^2_{S_2}(l[S_2]) &= \min\{\top, C^1_{S_1}(l[S_1]) \oplus U^b_2(l[x_2])\}\\
C^3_{S_2}(l[S_2]) &= \min\{\top, C^0_{S_1}(l[S_1]) \oplus U^b_2(l[x_2]), C^3_{S_1}(l[S_1]) \oplus U^a_2(l[x_2])\}\\
C^0_{S_3}(l[S_3]) &= \top\\
C^1_{S_3}(l[S_3]) &= \min\{\top, C^0_{S_2}(l[S_2]) \oplus U^a_3(l[x_3]), C^1_{S_2}(l[S_2]) \oplus U^a_3(l[x_3])\}\\
C^2_{S_3}(l[S_3]) &= \min\{\top, C^1_{S_2}(l[S_2]) \oplus U^b_3(l[x_3])\}\\
C^3_{S_3}(l[S_3]) &= \min\{\top, C^0_{S_2}(l[S_2]) \oplus U^b_3(l[x_3]), C^3_{S_2}(l[S_2]) \oplus U^a_3(l[x_3])\}\\
C_S(l) &= \min(C^2_{S_3}(l), C^3_{S_3}(l)) = \min(1,1) = 1
\end{aligned}$$

**Corollary 4.5.** *The* $\texttt{soft\_regular}^{var}$ *constraint is tractable projection-safe.*

*Proof.* Result follows from Theorem 3.12 and Theorem 4.4. □

Algorithm 4.2 computes the minimum cost of a $\texttt{soft\_regular}^{var}$ constraint. In the algorithm, it holds that $u^c_i = \min(\Delta(U^c_i))$ and $f^j_i = \min(\Delta(C^j_i))$. Line 1-2 computes the minimum cost of each unary constraint $U^c_i$. Since unary constraints are stored as tables, their minimum costs can be computed by scanning the tables. This step takes time $O(nd \cdot |\Sigma|)$. Line 3-4 initialize $f^j_0$ for all $q_j \in Q$

**Algorithm 4.2**: Compute the minimum cost of $\texttt{soft\_regular}^{var}$

**Input**: $C_S$: $\texttt{soft\_regular}^{var}(S, M)$, and a series of 1-projections/1-extensions $\Delta$

**Output**: $\min(\Delta(C_S))$

1 **for** $i \leftarrow 1$ **to** $n$ **do**
2  **for** $c \in \Sigma$ **do** $u_i^c \leftarrow \min(\Delta(U_i^c))$ ;
3 $f_0^0 \leftarrow 0$ ;
4 **for** $q_j \in Q \setminus \{q_0\}$ **do** $f_0^j \leftarrow \top$ ;
5 **for** $i \leftarrow 1$ **to** $n$ **do**
6  **for** $q_j \in Q$ **do**
7  $f_i^j = \min\{\top, \min_{((q_k,c)\mapsto q_j)\in T}\{f_i^k \oplus u_i^c\}\}$ ;
8 **return** $\min_{q_j \in F}\{f_n^j\}$ ;

following the definition. By definition of safe decomposition, and the decomposition shown in the proof of Theorem 4.4, we have:

$$\min(\Delta(C_i^j)) = \min\{\top, \min_{((q_k,c)\mapsto q_j)\in T}\{\min(\Delta(C_{i-1}^k)) \oplus \min(\Delta(U_i^c))\}\}$$

Line 5-7 of the algorithm computes the minimum costs accordingly, in time $O(n \cdot |T|)$. Finally, the algorithm returns the answer base on the fact that

$$\min(\Delta(C_S)) = \min_{q_j \in F}\{\min(\Delta(C_n^j))\}$$

This final step takes time $O(|F|)$. The total time consumed by the algorithm is $O(nd \cdot |M|)$.

**Theorem 4.6.** *Given a* $\texttt{soft\_regular}^{var}$ *constraint and a series of* 1*-projections/*1*-extensions* $\Delta$*. Algorithm 4.2 computes the minimum cost of* $\texttt{soft\_regular}^{var}$ *constraint after applying* $\Delta$*, in* $O(nd \cdot |M|)$ *time.*

## 4.3 Soft Grammar Constraint

$\texttt{soft\_grammar}^{var}$ is the soft form of the `grammar` constraint with the variable-based violation measure. In addition, when it is impossible to satisfy the underlying `grammar` constraint, the cost is $\top$. In the following we are to show

this constraint to be polynomially decomposable and thus tractable projection-safe, and give an algorithm to compute the minimum cost.

**Theorem 4.7.** *The* $\texttt{soft\_grammar}^{var}$ *constraint is polynomially decomposable.*

*Proof.* Let $C_S$ be the constraint $\texttt{soft\_grammar}^{var}(S, G)$ where $S = \{x_1, \ldots, x_n\}$ and $G = (\Sigma, N, P, T)$ is a context-free grammar in Chomsky Normal Form (see Definition 2.19). Let $S_{i,j} = \{x_i, \ldots, x_j\}$ for $i \leq j$. For each non-terminal $A \in N$, we define a context-free grammar $G_A = (\Sigma, N, P, A)$ obtained from $G$ by replacing the start non-terminal $T$ by $A$. We denote a constraint $\texttt{soft\_grammar}^{var}(S_{i,j}, G_A)$ by $C^A_{i,j}$. We also define the unary constraint $U^a_i$ on the variable $x_i$ where $a \in \Sigma$ that for all $v \in D(x_i)$:

$$U^a_i = \begin{cases} 0, & \text{if } v = a \\ 1, & \text{otherwise} \end{cases}$$

We show $C_S$ to be polynomially decomposable by constructing a sequence $[U^a_i, \ldots, C^A_{i,j}, \ldots, C_S]$. In the sequence $\{C^A_{i,j}\}$ are ordered in the increasing order of $i$. The length of the sequence is bounded by $O(n^2 \cdot |N| + nd)$. $C^A_{i,j}(v)$ for $i = j$ and $v \in D(x_i)$ can be determined base on all productions of the form $(A \mapsto a)$. We have

$$C^A_{i,i}(v) = \min\{\top, \min_{(A \mapsto a) \in P} U^a_i(v)\}$$

For $j > i$ and all assignments $l$ on $S_{i,j}$, we have

$$C^A_{i,j}(l) = \min\{\top, \min_{(A \mapsto A_1 A_2) \in P \wedge i \leq k < j} C^{A_1}_{i,k}(l[S_{i,k}]) \oplus C^{A_2}_{k+1,j}(l[S_{k+1,j}])\}$$

Finally, $C_S$ is equivalent to $C^T_{S_{1,n}}$. By Theorem 3.10, each constraint in the constructed sequence is either unary constraint, or can be safely decomposes into constraints precede it in the sequence. □

**Example 4.3.** *Suppose* $G = (\Sigma, N, P, T)$ *is a context-free grammar where* $\Sigma = \{a, b\}$, $N = \{T, A, B\}$ *and* $P$ *contains the following productions:*

$$T \mapsto AB,\ A \mapsto AA,\ B \mapsto BB,\ A \mapsto a,\ B \mapsto b$$

*$C_S$ is the constraint* $\texttt{soft\_grammar}^{var}(S, G)$ *where* $S = \{x_1, x_2, x_3\}$. *We construct the sequence*

$$\begin{aligned}
&[U_1^a, U_1^b, U_2^a, U_2^b, U_3^a, U_3^b,\\
&C_{1,1}^A, C_{1,1}^B, C_{1,1}^T, C_{2,2}^A, C_{2,2}^B, C_{2,2}^T, C_{3,3}^A, C_{3,3}^B, C_{3,3}^T,\\
&C_{1,2}^A, C_{1,2}^B, C_{1,2}^T, C_{2,3}^A, C_{2,3}^B, C_{2,3}^T\\
&C_{1,3}^A, C_{1,3}^B, C_{1,3}^T, C_S]
\end{aligned}$$

*For all assignments $l \in \mathcal{L}(S)$, we have:*

$$\begin{aligned}
C_{1,1}^A(l[S_{1,1}]) &= \min\{\top, U_1^a(l[x_1])\}\\
C_{1,1}^B(l[S_{1,1}]) &= \min\{\top, U_1^b(l[x_1])\}\\
C_{1,1}^T(l[S_{1,1}]) &= \top\\
&\dots\\
C_{1,2}^A(l[S_{1,2}]) &= \min\{\top, C_{1,1}^A(l[S_{1,1}]) \oplus C_{2,2}^A(l[S_{2,2}])\}\\
C_{1,2}^B(l[S_{1,2}]) &= \min\{\top, C_{1,1}^B(l[S_{1,1}]) \oplus C_{2,2}^B(l[S_{2,2}])\}\\
C_{1,2}^T(l[S_{1,2}]) &= \min\{\top, C_{1,1}^A(l[S_{1,1}]) \oplus C_{2,2}^B(l[S_{2,2}])\}\\
&\dots\\
C_{1,3}^A(l[S_{1,3}]) &= \min\{\top, C_{1,1}^A(l[S_{1,1}]) \oplus C_{2,3}^A(l[S_{2,3}]), C_{1,2}^A(l[S_{1,2}]) \oplus C_{3,3}^A(l[S_{3,3}])\}\\
C_{1,3}^B(l[S_{1,3}]) &= \min\{\top, C_{1,1}^B(l[S_{1,1}]) \oplus C_{2,3}^B(l[S_{2,3}]), C_{1,2}^B(l[S_{1,2}]) \oplus C_{3,3}^B(l[S_{3,3}])\}\\
C_{1,3}^T(l[S_{1,3}]) &= \min\{\top, C_{1,1}^A(l[S_{1,1}]) \oplus C_{2,3}^B(l[S_{2,3}]), C_{1,2}^A(l[S_{1,2}]) \oplus C_{3,3}^B(l[S_{3,3}])\}\\
C_S(l) &= C_{1,3}^T(l)
\end{aligned}$$

**Corollary 4.8.** *The* $\texttt{soft\_grammar}^{var}$ *constraint is tractable projection-safe.*

*Proof.* Result follows from Theorem 3.12 and Theorem 4.7. □

Algorithm 4.3 computes the minimum cost of a $\texttt{grammar}^{var}$ constraint after applying a series of 1-projections/1-extensions. In the algorithm, it holds that $u_i^c = \min(\Delta(U_i^c))$ and $f_{i,j}^A = \min(\Delta(C_{S_{i,j}}^A))$. Line 1-2 computes the minimum cost of each unary constraint $U_i^c$. As the unary constraints are stored as tables, the minimum costs can be computed by scanning the tables. This step takes

**Algorithm 4.3**: Compute the minimum cost of $\texttt{soft\_grammar}^{var}$

**Input**: $C_S$: $\texttt{soft\_grammar}^{var}(S, G)$, and a series of 1-projections/1-extensions $\Delta$
**Output**: $\min(\Delta(C_S))$

1 **for** $i \leftarrow 1$ **to** $n$ **do**
2  **for** $c \in \Sigma$ **do** $u_i^c \leftarrow \min(\Delta(U_i^c))$ ;
3 **for** $i \leftarrow 1$ **to** $n$ **do**
4  **for** $A \in N$ **do**
5  $f_{1,1}^A = \min\{\top, \min_{(A \mapsto a) \in P}\{u_i^a\}\}$ ;
6 **for** $len \leftarrow 2$ **to** $n$ **do**
7  **for** $i \leftarrow 1$ **to** $n - len + 1$ **do**
8  $j \leftarrow i + len - 1$ ;
9  **for** $A \in N$ **do**
10  $f_{i,j}^A \leftarrow \min\{\top, \min_{(A \mapsto A_1 A_2) \in P, i \le k < j}\{f_{i,k}^{A_1} \oplus f_{k+1,j}^{A_2}\}\}$ ;
11 **return** $f_{1,n}^T$ ;

time $O(n \cdot |\Sigma|)$. By definition of safe decomposition, and the decomposition shown in the proof in Theorem 4.7, we have

$$\min(\Delta(C_{S_{i,i}^A})) = \min\{\top, \min_{(A \mapsto a) \in P} \min(\Delta(U_i^a))\}$$
$$\min(\Delta(C_{S_{i,j}^A})) = \min\{\top, \min_{(A \mapsto A_1 A_2) \in P \wedge i \le k < j}\{\min(\Delta(C_{S_{i,k}^{A_1}})) \oplus \min(\Delta(C_{S_{k+1,j}^{A_2}}))\}\}$$

Line 3-10 of the algorithm computes the minimum costs accordingly, in $O(n^3 \cdot |P|)$ time. Finally, as $C_S$ is equivalent to $C_{S_{1,n}^T}^T$, the algorithm returns the value

$$\min(\Delta(C_S)) = \min(\Delta(C_{S_{1,n}^T}^T))$$

The total time consumed by the algorithm is $O((n^3 + nd) \cdot |G|)$.

**Theorem 4.9.** *Given a* $\texttt{soft\_grammar}^{var}$ *constraint and a series of* 1*-projections/*1*-extensions* $\Delta$*. Algorithm 4.3 computes the minimum cost of* $\texttt{soft\_grammar}^{var}$ *after applying* $\Delta$*, in* $O((n^3 + nd) \cdot |G|)$ *time.*

## 4.4 Max_Weight/Min_Weight Constraint

Given a set of variables $S$, a cost function $w(x_i, v)$ that maps a variable $x_i \in S$ and a value $v \in D_i$ to a cost $\in [0 \dots k]$. The `max_weight` constraint is the cost

function

$$\texttt{max\_weight}(S, w)(l) = \max_{x_i \in S \wedge l[x_i]=v} w(x_i, v)$$

The `min_weight` constraint is the cost function

$$\texttt{min\_weight}(S, w)(l) = \min_{x_i \in S \wedge l[x_i]=v} w(x_i, v)$$

These constraints are derived from the `maximum/minimum` constraint [5]. We are to show this constraint is polynomially decomposable and thus tractable projection-safe. We also give an algorithm to compute the minimum cost. Note that the following decomposition does not work. Let $C_S$ be the constraint $\texttt{min\_weight}(S, w)$. For all $l \in \mathcal{L}(S)$, we have

$$C_S(l) = \min_{1 \le i \le n} \{C_i(l[x_i])\}$$

where each $C_i$ is the unary constraint such that

$$C_i(v) = w(i, v)$$

holds for all $v \in D(x_i)$. The decomposition is not a safe decomposition since the scope of $C_i$ is not $S$, thus Theorem 3.8 does not applies.

**Theorem 4.10.** *The* $\texttt{max\_weight}(S, w)$ *and* $\texttt{min\_weight}(S, w)$ *constraints are polynomially decomposable.*

*Proof.* We prove for `max_weight`. The proof for `min_weight` is similar. Let $C_S$ be the constraint $\texttt{max\_weight}(S, w)$ where $S = \{x_1, \ldots, x_n\}$. We define two sets of unary constraints $\{H_i^u\}$ and $\{G_i^\alpha\}$ as follows:

$$H_i^u(v) = \begin{cases} w(i, v), & v = u \\ \top, & v \neq u \end{cases}$$

$$G_i^\alpha(v) = \begin{cases} 0, & w(x_i, v) \le \alpha \\ \top, & w(x_i, v) > w(x_j, u) \end{cases}$$

where $1 \leq i \leq n$, $u \in D(x_i)$ and $\alpha$ is a cost. We construct the sequence $[H_i^u, \ldots, G_j^v, \ldots, C_S]$ by enumerating all $1 \leq i, j \leq n, u \in D(x_i)$ and $\alpha \in \{w(k,v) | 1 \leq k \leq n \wedge v \in D(x_k)\}$ . The length of the sequence is bounded by $O(n^2 d)$. The decomposition is gave as follows:

$$C_S(l) = \min_{1 \leq i \leq n \wedge v \in D(x_i)} \{H_i^v(l[x_i]) \oplus \bigoplus_{x_j \in S, i \neq j} G_j^{w(i,v)}(l[x_j])\}$$

In the equation $H_i^v$ represents the choice of the maximum weighted component in the tuple, $G_i^{j,u}$ represents the choice of each components other than the maximum weighted one. By Lemma 3.10, the decomposition is a safe decomposition. □

**Example 4.4.** *Suppose $C_S$ is the constraint* `max_weight`$(S, w)$ *where $S = \{x_1, x_2\}$. $D(x_1) = \{1, 4\}$, $D(x_2) = \{2, 3\}$. $H_i^u$ and $G_i^u$ are defined as above. We construct the sequence*

$$[H_1^1, H_1^4, H_2^2, H_2^3, G_1^1, G_1^4, G_2^2, G_2^3, C_S]$$

*Then for all assignments $l \in \mathcal{L}(S)$, we have:*

$$C_S(l) = \min \begin{cases} H_1^4(l[x_1]) \oplus G_2^3(l[x_2]) \\ G_1^1(l[x_1]) \oplus G_2^2(l[x_2]) \\ G_1^1(l[x_1]) \oplus G_2^3(l[x_3]) \end{cases}$$

**Corollary 4.11.** `max_weight` *and* `min_weight` *constraints are tractable projection-safe.*

*Proof.* Result follows from Theorem 3.12 and Theorem 4.10. □

Here we give an $O(nd \log(nd))$ algorithm to compute the minimum cost of a `max_weight` constraint after a series of 1-projections/1-extensions. The algorithm first organize all the variable-value pairs $(x_i, v)$ where $x_i \in S, v \in D(x_i)$ in an array $A$, and sort them by $w(x_i, v)$. Then it scans $A$ (Line 6-11).

**Algorithm 4.4**: Computing the minimum cost of `max_weight`

**Input**: $C_S$: `max_weight`$(S, w)$
**Output**: a set $\{m_{i,v}\}$, where $m_{i,v} = \min\{C_S(l) | l[x_i] = v\}$

```
1  A ← an array of pair (x_j, u) for all possible x_j ∈ X, u ∈ D_j ;
2  sort A = {(x_j, u))} in increasing order of w(x_j, u) ;
3  for i ← 1 to n do  a_i ← ⊤ ;
4  s ← n · ⊤ ;
5  r ← ⊤ ;
6  for k ← 1 to |A| do
7      (x_i, v) ← A[k] ;
8      s ← s − a_i ;
9      a_i ← min(a_i, Δ(G_i^v(v))) ;
10     s ← s + a_i ;
11     r ← min(r, Δ(H_i^v(v)) + s − a_i) ;
12 return r ;
```

At the $k$-th iteration where $A[k] = (x_i, v)$, it maintains that $a_i = \min(G_i^v)$, and $s = \sum_{1 \le i \le n} a_i$ (Line 9-10). Thus,

$$\min(H_i^v) \oplus \bigoplus_{x_j \in S \setminus \{x_i\}} \min(G_j^{prev_j(w(i,v))}) = \min\{\top, s - a_i + \Delta(H_i^v(v))\}$$

So at the end of the algorithm, it holds that

$$r = \min_{1 \le i \le n \wedge v \in D(x_i)} \{\min(\Delta(H_i^v)) \oplus \bigoplus_{x_j \in S \setminus \{x_i\}} \min(\Delta(G_j^{w(i,v)}))\} = \min(\Delta(C_S))$$

**Theorem 4.12.** *Given a* `max_weight` *constraint and a series of* 1*-projections/*1*-extensions* $\Delta$*. Algorithm 4.4 computes the minimum cost of the* `max_weight` *constraint after applying* $\Delta$*, in* $O(nd \cdot \log(nd))$ *time.*

# Chapter 5

# Experiments

We implement the constraints described in the previous chapter in Toulbar2 v0.9 to demonstrate the practicality of our algorithmic framework. We also compare the results with different levels of consistency: strong ∅IC, GAC* and FDGAC*, which cover 0-projection, 1-projection, 1-extensions. For each constraint discussed in Chapter 4, we conduct one experiment to demonstrate the efficiency.

In the experiments, variables are assigned in lexicographic order. Value assignment starts with the values with minimum unary cost. The tests are conducted on an Intel Core2 Duo E7400 (2 x 2.80GHz) machine with 4GB RAM. Each benchmark has a different timeout. We first compare the number of solved instances. Among those solved instances, we report their average run-time and number of backtracks. Out of 10 randomly generated test cases fo each parameter setting, the best result is highlighted in bold. All the benchmark problems are NP-Hard.

## 5.1 The car Sequencing Problem

The car sequencing problem (prob001 in CSPLib) [38] Given $n$ cars of different type. Each type is specified by a set of options. For the $i$-th option, for every $c_i$ cars in the assembly line, the maximum number of cars allowed with option

$i$ is $m_i$. This problem is to find a production sequence to satisfy all above constraints. We use $n$ variables with domain 1 to $n$ to model this problem. The variable $x_i$ denotes the type of the $i$-th car in the sequence. One *gcc* constraint ensures all cars needed lie in the sequences. For the $i$-th option, $n - m_i + 1$ **among** constraints ensure the capacity on each option. We generate 10 over-constrained instances randomly, each with 5 options, and block size of at most 5, i.e. $1 \leq m \leq c_i \leq 5$. Each car is randomly assigned to a type and each type has 1/2 chance to have an option. To softened the problem, we replace **among** constraints by $\texttt{soft\_among}^{var}$. We also model the hard gcc constraints as $\texttt{soft\_gcc}^{var}$ [49] which returns $\top$ upon violation.

Results are shown in Table 5.1. In this benchmark, solving with FDGAC* runs faster than GAC* up to nearly 5 times, and more than 20 times faster than strong ∅IC. Enforcing GAC* also speeds up by 6 times when compared with strong ∅IC.

| $n$ | strong ∅IC | | | GAC* | | | FDGAC* | | |
|---|---|---|---|---|---|---|---|---|---|
| | solved | time | backtracks | solved | time | backtracks | solved | time | backtracks |
| 14 | 10 | 42.84 | 234537 | 10 | 16.80 | 67842 | **10** | **1.37** | **1607** |
| 15 | 8 | 136 | 715754 | 10 | 29.75 | 109085 | **10** | **4.49** | **4978** |
| 16 | 3 | 178.98 | 834998 | 8 | 133.08 | 434969 | **10** | **6.90** | **6179** |
| 17 | 1 | 163.73 | 830343 | 2 | 130.14 | 387446 | **10** | **48.07** | **35218** |

Table 5.1: The number of solved instances (in 5 minutes time limit), the average time (in seconds) of solved instances and the number of backtracks in solving the car sequencing problem using $\texttt{soft\_among}^{var}$ constraints

## 5.2 The nonogram problem

Nonogram problem (prob012 in CSPLib) [6] Given a board of size $n \times n$. The problem is to find a black-white coloring on each cell such that each row and each column contain a specific set of sequences of black squares with different lengths. For example, we can specify that a row must have two consecutive black blocks, one with length 2 and the other with length 3. We

model the problem by $n^2$ variables, among which $x_{ij}$ denotes the color of the block at the $i$-th row and $j$-th column. We model the restrictions on each row and column by a `regular` constraint. To soften the problem, we replace `regular` constraints by `soft_regular`$^{var}$. We generate random instances for the problem by generating a set of sequences for each row and column.

In a time limit of 5 minutes, enforcing strong ∅IC can only solve relatively small instances ($n = 6$). Enforcing GAC* can solve larger ones ($n = 8$). For $n = 10$, all instances can be solved only when FDGAC* is enforced, where each instance is solved in around 10 seconds on average. The `regular`$^{var}$ constraint is both polynomially decomposable and (flow-based) projection-safe [30, 32]. So we compare the two approaches. The results are shown in Table 5.2. The two approaches would result in the same search tree when we enforce the same consistency, but the run-time varies. Our benchmarks show that for `regular` constraints, the polynomially decomposable approach is more efficient than the flow-based approach. It is because the constant factor behind the flow algorithm is usually large.

| polynomially decomposable | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $n$ | strong ∅IC | | | GAC* | | | FDGAC* | | |
| | solved | time | backtrack | solved | time | backtrack | solved | time | backtrack |
| 6 | 10 | 9.50 | 150167 | 10 | 0.03 | 763 | **10** | **0.00** | **109** |
| 7 | 1 | 245.17 | 2627322 | 10 | 3.88 | 72811 | **10** | **0.03** | **345** |
| 8 | 0 | * | * | 7 | 113.76 | 1730882 | **10** | **0.12** | **842** |
| 9 | 0 | * | * | 2 | 52.85 | 764467 | **10** | **0.34** | **1500** |
| 10 | 0 | * | * | 0 | * | * | **10** | **11.78** | **22828** |
| flow-based approach | | | | | | | | | |
| $n$ | strong ∅IC | | | GAC* | | | FDGAC* | | |
| | solved | time | backtrack | solved | time | backtrack | solved | time | backtrack |
| 6 | 9 | 25.23 | 72130 | 10 | 0.32 | 763 | 10 | 0.05 | 109 |
| 7 | 0 | * | * | 10 | 60.84 | 72811 | 10 | 0.48 | 345 |
| 8 | 0 | * | * | 1 | 26.59 | 28166 | 10 | 2.04 | 842 |
| 9 | 0 | * | * | 1 | 151.38 | 83479 | 10 | 5.87 | 1500 |
| 10 | 0 | * | * | 0 | * | * | 9 | 40.67 | 4848 |

Table 5.2: The number of solved instances (in 5 minutes time limit), the average time (in seconds) of solved instances and the number of backtracks in solving the nonogram problem using `soft_regular`$^{var}$ constraints

## 5.3 Well-Formed Parenthesis

Given a set of $n$ even length intervals in $[1, \ldots, n]$, where $n$ is an even number. The problem is to find a string of parentheses of length $n$, such that substrings in each of the intervals are well-formed parentheses. We model this problem by a set of $n$ variables. For each interval, a `soft_grammar` constraint is posted to represent the well-formed parentheses requirement. The problem is softened by associating variable-based violation measure to each `grammar` constraint. We generate $n-1$ even length intervals by randomly picking their end points in $[1, \ldots, n]$, and add an interval covering the whole range to ensure that all variables are constrained. We also randomly assign unary costs to the variables. As shown in Table 5.3, FDGAC* is up to one order of magnitude faster than strong ∅IC, and up to 4 times faster than GAC*.

| $n$ | strong ∅IC | | | GAC* | | | FDGAC* | | |
|---|---|---|---|---|---|---|---|---|---|
| | solved | time | backtrack | solved | time | backtrack | solved | time | backtrack |
| 20 | 10 | 6.36 | 5552 | 10 | 0.54 | 408 | **10** | **0.43** | **172** |
| 22 | 10 | 17.38 | 10253 | 10 | 1.48 | 784 | **10** | **0.92** | **245** |
| 24 | 10 | 47.19 | 22668 | 10 | 3.38 | 1383 | **10** | **2.08** | **394** |
| 26 | 9 | 90.94 | 34435 | 10 | 6.98 | 2175 | **10** | **2.89** | **440** |
| 28 | 4 | 176.1 | 59756 | 10 | 31.99 | 7208 | **10** | **7.75** | **765** |
| 30 | 0 | * | * | 10 | 56.43 | 9705 | **10** | **15.59** | **1026** |
| 32 | 0 | * | * | 10 | 85.58 | 14825 | **10** | **20.12** | **1367** |
| 34 | 0 | * | * | 6 | 158.16 | 25546 | **10** | **54.94** | **3346** |

Table 5.3: The number of solved instances (in 5 minutes time limit), the average time (in seconds) of solved instances and the number of backtracks in solving the well-formed parenthesis problem using $\texttt{soft\_grammar}^{var}$ constraints

## 5.4 Minimum Energy Broadcasting Problem

The minimum energy broadcasting problem (prob048 in CSPLib) [15] Given $n$ wireless routers in the network, one of which is the root that broadcasts messages to every other router. Not all links between pairs of routers are available, and each available link requires an energy level $e_{ij}$. The energy

consumed by a router is equal to the maximum energy among all the links required to send the messages. The task is to find a broadcast tree that minimizes the total energy consumed. We use $n$ variables, where $x_i$ denotes the index of the router from which the $i$-th router receive a message. In addition, $j \in D(x_i)$ iff there is a link between the $i$-th and $j$-th routers. One hard global constraint `tree` [8] is posted to enforce an assignment representing a tree. We post $n$ `max_weight` cost functions to represent the energy consumed by each router. For the $i$-th router, we post a constraint $\texttt{max\_weight}(X, w_i)$, where $X$ is the set of all variables, and $w_i(x_j, k) = e_{ij}$ if $k = i$, or 0 otherwise. We randomly generate 10 instances of randomly connected network for each configuration of $n$ routers and $m$ links. Links are uniformly distributed between all pair of routers with a random energy requirement. GAC is enforced on the `tree` constraint.

In this benchmark, however, we get a result different from the previous ones. GAC* benefits from its pruning power and speeds up the solving by around 2 times compared to strong ∅IC. Although FDGAC* can reduce more search spaces than GAC* up to 6 times, the run-time is worse than GAC* by 2 times. We notice that in our model, the scope of each constraint involves all variables. Whenever a unary constraint increases cost, consistency checking is invoked for all global constraints, which introduces a large overhead. We also notice that the hard `tree` global constraint achieves strong ∅IC and GAC* (because it is a hard constraint) but not FDGAC*, which could also be the explanation of the result.

| $n$ | $m$ | strong ∅IC | | | GAC* | | | FDGAC* | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | solved | time | backtrack | solved | time | backtrack | solved | time | backtrack |
| 20 | 40 | 10 | 8.03 | 61806 | **10** | **1.64** | 9080 | 10 | 2.03 | **1352** |
| 20 | 60 | 10 | 26.08 | 153237 | **10** | **13.54** | 55317 | 10 | 37.77 | **16694** |
| 20 | 100 | 10 | 13.55 | 69453 | **10** | **12.50** | 37323 | 10 | 41.78 | **12106** |
| 25 | 50 | 10 | 72.55 | 303422 | **10** | **15.34** | 52855 | 10 | 15.48 | **4849** |
| 25 | 75 | 5 | 301.68 | 1044058 | **7** | **229.10** | 625415 | 5 | 176.45 | **34108** |
| 25 | 125 | 5 | 50.27 | 121473 | **5** | **43.04** | 73262 | 3 | 166.85 | **22005** |
| 30 | 60 | 4 | 216.44 | 557575 | **9** | **101.33** | 233610 | 9 | 118.48 | **21424** |
| 30 | 90 | 1 | 401.92 | 1050414 | **2** | **162.63** | 293660 | 1 | 305.96 | **43238** |

Table 5.4: The number of solved instances (in 10 minutes time limit), the average time (in seconds) of solved instances and the number of backtracks in solving the minimum energy broadcasting problem using `max_weight` constraints

# Chapter 6

# Related Work

In this chapter, we present researches that related to our work. We briefly describe various consistency techniques in WCSP, and related works on global constraints.

## 6.1 WCSP Consistencies

The WCSP framework is useful in modeling many over constrained problems and optimization problems. To solve WCSP efficiently, many consistency techniques have been proposed. NC* and AC* were developed by Larrosa and Schiex [28]. They demonstrated a branch-and-bound algorithm that maintains AC*. Other forms of consistency notions with different pruning power appeared later, including FDAC* [27], EDAC* [19], $\varnothing$-IC [51] and strong $\varnothing$-IC [30]. Cooper *et al.* [17] defined two consistency notions, namely OSAC and VAC, both of which require a relaxation of cost valuation structure $V(\top)$ to real numbers. $k$-consistency is due to Cooper [16].

AC*, FDAC* and EDAC* are specialized for binary constraints, yet they can be generalized to handle high arity constraints and global constraints. Generalized version of arc consistency star, GAC*, is defined by Cooper and Schiex [18]. Sanchez *et al.* [47] extended AC*, FDAC* and EDAC* for ternary constraints. Their method to enforce EDAC* on ternary constraint requires

2-extension. FDGAC* is due to Lee and Leung [30, 32]. They also showed that naively generalizing EDAC* to high arity constraints will lead to an oscillation in the enforcement algorithm, and proposed a weak form of EDGAC* based on cost providing partition [31, 32].

Bound arc consistency (BAC*) is discussed by Zytnicki *et al.* [51] to handle WCSP with large domains. This consistency notion only consider domain bounds, and only require simple supports for boundary values in the variable domains.

Different consistency notions depend on different projections/extensions. For example, 0-projections/0-extensions are employed in ∅IC and strong ∅IC enforcements. 1-projections/1-extensions are the backbone of the consistency algorithms of (G)AC*, FD(G)AC* and (weak) ED(G)AC*. Projections and extensions of arity larger than 1 are required for enforcing EDAC* in ternary constraints and $k$-consistency. Our work show that we can efficiently applied those consistencies depends on 1-projections/1-extensions to polynomially decomposable constraints, and those consistencies depends on 2-projections/2-extensions or above are hard to enforce efficiently on global constraints.

## 6.2 Global Constraints

Global constraint is one key element to make CSP framework success. A global constraint could be understood as an *expressive and concise condition involving a non-fixed number of variables* [6]. Since the work of Lauriere on ALICE [29], many global constraints have been proposed and studied. Famous examples include the `allDifferent` constraint [29] and `cumulative` constraint [1]. On the other hand, early work on WCSP consistencies concentrated on binary table constraints, and recently generalized to handle high arity constraints and global constraints. Introduction of global constraints to WCSP is a must to make the WCSP framework more useful. In this section, we review related

works on global constraints discussed in our work, and constraint softening.

The `among` constraint was originally proposed by Beldiceanu and Contejean [7]. An algorithm to achieve AC was given by Bessière *et al.*[11]. This constraint is useful in modeling the car-sequencing problem [38].

The `regular` constraint is proposed by Peasant [40]. This constraint is extremely useful since it is able to model many other global constraints, including `stretch` and `pattern`. He also gave an algorithm that achieves AC on the `regular` constraint based on a layered directed graph representation of the constraint. The decomposition for the `soft_regular` constraint is derived from such a graph representation.

The `grammar` constraint is proposed by Kadioglu and Sellmann [22], and Quimper and Walsh [42]. A CYK parser based algorithm [22, 42] and an Earley parser based algorithm [42] are given to achieve AC on this constraint. The CYK parser based AC algorithm is improved by Kadioglu and Sellmann [21]. They also consider achieving AC when `grammar` constraint appears in conjunction with a linear objective function. Quimper and Walsh discussed decomposition of the `grammar` constraint [43, 44]. Kassirelos *et al.* proposed the `weightedGrammar` constraint [24], which can be used to model $\texttt{soft\_grammar}^{var}$ and $\texttt{soft\_grammar}^{edit}$ constraints. Restricted classes of the `grammar` constraint was also discussed in literature [23].

The `max_weight/min_weight` constraints are derived from the `maximum/minimum` constraints originated in CHIP [7]. Beldiceanu showed that the two constraints are instances of the minimum constraint family, and presented a filtering algorithm.

Constraint softening is proposed by Règin *et al.* [45] to model and solve over-constrained problems. Van Hoeve *et al.* [49] make use of flow theory to compute the minimum cost of several soft constraints, including soft variants of the `allDifferent`, `gcc` and `regular` constraints. Lee and Leung [30, 32]

further extend their idea and show these constraints are flow-based projection-safe.

# Chapter 7

# Conclusion

In this section, we summarize our contributions and shed light on possible future directions of our research.

## 7.1 Contributions

In this thesis, we discuss tractable projection-safety, and introduce the concept of polynomially decomposable constraints. Our contributions are three-fold.

First, we address the issue of *tractable projection-safety* in enforcing WCSP consistencies. WCSP consistencies an be efficiently enforced only when tractable projection-safety is guaranteed. We divide our discussion into three cases of different scenarios of projections and extensions. We show that projection-safety is always possible for projections/extension to/from the nullary constraint, while it is alway impossible for projections/extensions to/from $r$-ary constraints for $r \geq 2$. When $r = 1$, we show that a tractable constraint may or may not be tractable projection-safe by giving positive and negative examples.

Second, we define *polynomially decomposable soft constraints* based on *safe decomposition*. Safe decomposition divides a soft constraint into sub-constraints which allows us to (1) compute the minimum cost of the original constraint from the minimum cost of its sub-constraints, and (2) distribute

projections and extensions to its sub-constraints. We give special scenarios of safe decomposition. We show that a polynomially decomposable soft constraints are tractable and tractable projection-safe, since we can apply a dynamic programming approach to compute the minimum cost of such a soft constraint, and the approach is still applicable after projections and extensions. We further show that $\texttt{soft\_among}^{var}$, $\texttt{soft\_regular}^{var}$, $\texttt{soft\_grammar}^{var}$ and `max_weight/min_weight` constraints are polynomially decomposable. We give the decomposition of these constraint, and base on the decomposition, we give algorithms to compute their minimum costs. Our effort give rise to another class of tractable projection-safe soft global constraint.

Third, we perform experiments and compare typical WCSP consistency notions and show that our algorithm framework works well with GAC* and FDGAC* enforcement both, in terms of run-time and reduction in search space. We also compare our approach with the flow-based approach [30]. We show that our approach is more competitive.

## 7.2 Future Work

We have discussed the issue of tractable projection-safety in WCSP consistencies enforcement, and show that flow-based projection-safe constraints and polynomially decomposable constraints are tractable projection-safe. An immediate future work is to investigate other forms of tractable projection-safety and techniques for enforcing typical consistencies efficiently.

The second possible research question is whether we can handle constraints that are intractable efficiently. Existing consistency techniques for WCSP requires knowledge of the minimum costs of constraints in the system. The question is whether we can design weak forms of these consistency techniques that requires knowledge of a lower bound of the minimum costs, which allows application of approximation algorithms to compute minimum costs of

intractable constraints.

The third possible direction is related to optimal soft arc consistency (OSAC) [17], where a sequence of 1-projections/1-extensions operation which yields an optimal $C_\varnothing$ is identified. Such task can be done by solving a linear program. OSAC is specialize for table constraints, and we would like to extend the idea to global constraints. The difficulty lies in how we can post linear constraints in the linear program to require a global constraint in the WCSP to have non-negative minimum cost. We conjecture that the decompositions given in Chapter 4 is related to such a reformulation.

# Bibliography


[1] AGGOUN, A., AND BELDICEANU, N. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling 17*, 7 (1993), 57–73.

[2] APT, K. The essence of constraint propagation. *Theoretical computer science 221*, 1-2 (1999), 179–210.

[3] APT, K. The rough guide to constraint propagation. In *Proceedings of 5th International Conference on Principles and Practice of Constraint Programming* (1999), Springer, pp. 1–23.

[4] APT, K. *Principles of constraint programming*. Cambridge Univ Pr, 2003.

[5] BELDICEANU, N. Pruning for the minimum constraint family and for the number of distinct values constraint family. In *Proceedings of 7th International Conference on Principles and Practice of Constraint Programming* (2001), pp. 211–224.

[6] BELDICEANU, N., CARLSSON, M., AND RAMPON, J. Global constraint catalog. *SICS Research Report* (2005).

[7] BELDICEANU, N., AND CONTEJEAN, E. Introducing global constraints in chip. *Mathematical and Computer Modelling 20*, 12 (1994), 97–123.

[8] BELDICEANU, N., FLENER, P., AND LORCA, X. The tree constraint. In *Proceedings of 2nd Integration of AI and OR Techniques in Constraint*

*Programming for Combinatorial Optimization Problems* (2005), Springer, pp. 64–78.

[9] BESSIÈRE, C. Arc-consistency and arc-consistency again. *Artificial intelligence 65*, 1 (1994), 179–190.

[10] BESSIÈRE, C., FREUDER, E., AND RÉGIN, J. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence 107*, 1 (1999), 125–148.

[11] BESSIERE, C., HEBRARD, E., HNICH, B., KIZILTAN, Z., AND WALSH, T. Among, common and disjoint constraints. *Recent Advances in Constraints* (2006), 29–43.

[12] BESSIERE, C., HEBRARD, E., HNICH, B., AND WALSH, T. The complexity of global constraints. In *Proceedings of 19th AAAI Conference on Artificial Intelligence* (2004), pp. 112–117.

[13] BESSIÈRE, C., AND RÉGIN, J. Refining the basic constraint propagation algorithm. In *Proceedings of 17th International Joint Conferences on Artificial Intelligence* (2001), vol. 17, Citeseer, pp. 309–315.

[14] BESSIÈRE, C., RÉGIN, J., YAP, R., AND ZHANG, Y. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence 165*, 2 (2005), 165–185.

[15] BURKE, D., AND BROWN, K. Using relaxations to improve search in distributed constraint optimisation. *Artificial Intelligence Review 28*, 1 (2007), 35–50.

[16] COOPER, M. High-order consistency in valued constraint satisfaction. *Constraints 10*, 3 (2005), 283–305.

[17] COOPER, M., DE GIVRY, S., SANCHEZ, M., SCHIEX, T., ZYTNICKI, M., AND WERNER, T. Soft arc consistency revisited. *Artificial Intelligence 174*, 7-8 (2010), 449–478.

[18] COOPER, M., AND SCHIEX, T. Arc consistency for soft constraints. *Artificial Intelligence 154*, 1-2 (2004), 199–227.

[19] DE GIVRY, S., HERAS, F., ZYTNICKI, M., AND LARROSA, J. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *Proceedings of 19th International Joint Conferences on Artificial Intelligence* (2005), pp. 84–89.

[20] FLUM, J., AND GROHE, M. *Parameterized complexity theory*. Springer-Verlag New York Inc, 2006.

[21] KADIOGLU, S., AND SELLMANN, M. Efficient context-free grammar constraints. In *Proceedings of 23rd AAAI Conference on Artificial Intelligence* (2008), pp. 310–316.

[22] KADIOGLU, S., AND SELLMANN, M. Grammar constraints. *Constraints 15*, 1 (2010), 117–144.

[23] KATSIRELOS, G., MANETH, S., NARODYTSKA, N., AND WALSH, T. Restricted global grammar constraints. In *Proceedings of 15th International Conference on Principles and Practice of Constraint Programming* (2009), Springer, pp. 501–508.

[24] KATSIRELOS, G., NARODYTSKA, N., AND WALSH, T. The weighted Grammar constraint. *Annals of Operations Research 184* (2011), 179–207.

[25] KROM, M. The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary. *Mathematical Logic Quarterly 13*, 1-2 (1967), 15–20.

[26] LAND, A., AND DOIG, A. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society* (1960), 497–520.

[27] LARROSA, J., AND SCHIEX, T. In the quest of the best form of local consistency for weighted CSP. In *Proceedings of 18th International Joint Conferences on Artificial Intelligence* (2003), pp. 239–244.

[28] LARROSA, J., AND SCHIEX, T. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence 159*, 1-2 (2004), 1–26.

[29] LAURIERE, J. A language and a program for stating and solving combinatorial problems. *Artificial intelligence 10*, 1 (1978), 29–127.

[30] LEE, J., AND LEUNG, K. Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In *Proceedings of 21st International Joint Conferences on Artificial Intelligence* (2009), pp. 559–565.

[31] LEE, J., AND LEUNG, K. A Stronger Consistency for Soft Global Constraints in Weighted Constraint Satisfaction. In *Proceedings of 24th AAAI Conference on Artificial Intelligence* (2010).

[32] LEUNG, K. L. Soft global constraints in constraint optimization and weight constraint statisfaction. Master's thesis, The Chinese University of Hong Kong, 2009.

[33] MACKWORTH, A. Consistency in networks of relations. *Artificial intelligence 8*, 1 (1977), 99–118.

[34] MACKWORTH, A. Consistency in networks of relations. *Artificial intelligence 8*, 1 (1977), 99–118.

[35] MOHR, R., AND HENDERSON, T. Arc and path consistency revisited. *Artificial Intelligence 28*, 2 (1986), 225–233.

[36] MOHR, R., AND MASINI, G. Good old discrete relaxation. In *8th European Conference on Artificial Intelligence* (1988), pp. 651–656.

[37] MONTANARI, U. Networks of constraints: Fundamental properties and applications to picture processing. *Information sciences 7* (1974), 95–132.

[38] PARRELLO, B., KABAT, W., AND WOS, L. Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *Journal of Automated Reasoning 2*, 1 (1986), 1–42.

[39] PERLIN, M. Arc consistency for factorable relations. *Artificial Intelligence 53*, 2-3 (1992), 329–342.

[40] PESANT, G. A regular language membership constraint for finite sequences of variables. In *Proceedings of 10th Conference on Principles and Practice of Constraint Programming* (2004), pp. 482–495.

[41] PETIT, T., RÉGIN, J., AND BESSIÈRE, C. Specific filtering algorithms for over-constrained problems. In *Proceedings of 7th International Conference on Principles and Practice of Constraint Programming* (2001), Springer, pp. 451–463.

[42] QUIMPER, C., AND WALSH, T. Global grammar constraints. In *Proceedings of 12th International Conference on Principles and Practice of Constraint Programming* (2006), pp. 751–755.

[43] QUIMPER, C., AND WALSH, T. Decomposing global grammar constraints. In *Proceedings of 13th International Conference on Principles and Practice of Constraint Programming* (2007), Springer, pp. 590–604.

[44] QUIMPER, C., AND WALSH, T. Decompositions of grammar constraints. In *Proceedings of 23rd AAAI Conference on Artificial Intelligence* (2008), pp. 1567–1570.

[45] RÉGIN, J., PETIT, T., BESSIÈRE, C., AND PUGET, J. An original constraint based approach for solving over constrained problems. In *Proceedings of 6th International Conference on Principles and Practice of Constraint Programming* (2000), Springer, pp. 543–548.

[46] SABIN, D., AND FREUDER, E. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 2nd International Workshop on Principle and Practice of Constraint Programming* (1994), Springer, pp. 10–20.

[47] SANCHEZ, M., DE GIVRY, S., AND SCHIEX, T. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints 13*, 1 (2008), 130–154.

[48] SCHIEX, T., FARGIER, H., AND VERFAILLIE, G. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of 14th International Joint Conferences on Artificial Intelligence* (1995), pp. 631–639.

[49] VAN HOEVE, W., PESANT, G., AND ROUSSEAU, L. On global warming: Flow-based soft global constraints. *Journal of Heuristics 12*, 4 (2006), 347–373.

[50] ZHANG, Y., AND YAP, R. Making ac-3 an optimal algorithm. In *Proceedings of 17th International Joint Conferences on Artificial Intelligence* (2001), pp. 316–321.

[51] Zytnicki, M., Gaspin, C., De Givry, S., and Schiex, T. Bounds arc consistency for weighted csps. *Journal of Artificial Intelligence Research 35*, 1 (2009), 593–621.