Consistency Techniques for Linear Global Cost Functions in Weighted Constraint Satisfaction

SHUM, Yu Wai

A Thesis Submitted in Partial Fulfilment

of the Requirements for the Degree of

Master of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

August 2012

Thesis/Assessment Committee

Prof. WONG Tien Tsin (Chair)
Prof. LEE Ho Man Jimmy (Thesis Supervisor)
Prof. LEE Pak Ching (Committee Member)
Prof. Javier LARROSA (External Examiner)

Abstract

The solving of Weighted CSP (WCSP) with global cost functions relies on powerful consistency techniques, but enforcing these consistencies on global cost functions is not a trivial task. Lee and Leung suggest that a global cost function can be used practically if we can find its minimum cost and perform projections/extensions on it in polynomial time, and at the same time projections and extensions should not destroy those conditions. However, there are many useful cost functions with no known polynomial time algorithms to compute the minimum costs yet.

We propose a special class of global cost functions which can be modeled as integer linear programs, called polynomially linear projection-safe (PLPS) cost functions. We show that their minimum cost can be computed by integer programming and this property is unaffected by projections/extensions. By linear relaxation we can avoid the possible NP-hard time taken to solve the integer programs, as the approximation of their actual minimum costs can be obtained to serve as a good lower bound in enforcing the relaxed forms of common consistencies.

We show the benefits of using the conjunctions of PLPS cost functions empirically in terms of runtime. We introduce integral polynomially linear projection-safe (IPLPS) cost functions as a subclass of PLPS cost functions whose allow us to characterize the benefits of using the conjunctions of them. Given a standard WCSP consistency α, we give theorems showing that maintaining relaxed α on a conjunction of IPLPS cost functions is stronger than maintaining α on the individual cost functions. A useful application of our method is on some IPLPS global cost functions, whose minimum cost computations are tractable and yet those for their

conjunctions are not. We show that an important subclass of flow-based projection-safe and polynomially decomposable cost functions falls into this category.

Experiments are conducted to demonstrate the feasibility and efficiency of our framework. We observe orders of magnitude in runtime and search space improvements by using the conjunctions of PLPS and IPLPS cost functions with relaxed consistencies when compared with the existing approaches.

摘要

在加權約束滿足問題中使用多元價值函數需要强大的一致相容性技術，而在多元價值函數中維護一致相容性並不是一項簡單的工作。能在多項式時間內找出多元價值函數的最少價值，而且不被投影及擴展操作所破壞，是讓該多元價值函數實用的主要條件。但是，有很多有用的多元價值函數尚未有多項式時間的算法找出其最少價值，因而未能在加權約束滿足問題中實用地使用它們。

我們定義了一類可被建構為整數線性規劃的多元價值函數，並稱它們為多項式線性投影安全(PLPS)價值函數。該類價值函數的最少價值能由解答整數線性規劃中找出，而這個特性並不會被投影及擴展操作所影響。線性鬆馳能讓我們找出一個最少價值的接近值，並避免了解答整數線性規劃的NP-難困難性。該最少價值的接近值能作為最少價值的下限以供維護鬆馳一致相容性概念。

在實踐中我們示範了使用PLPS價值函數的組合的好處。我們定義了整數多項式線性投影安全(IPLPS)價值函數作為PLPS價值函數的一個子類，並讓我們表示組合該類價值函數的好處。在一個加權約束滿足問題的一致相容性α中，我們表示了在IPLPS價值函數的組合中維護鬆馳α比在單獨的IPLPS價值函數中維護α强大。這結果可用在能在多項式時間中找出最少價值，但不能在多項式時間中找出它們的組合的最少價值的IPLPS價值函數中。基於流量投影安全(flow-based projection-safe)及可多項式分解(polynomially decomposable)價值函數的一個重要的子類屬於這一類的IPLPS價值函數。

在實驗中我們展示了我們的方法的可行性和效率。無論在時間或搜索空

間的改進上，與現有的方法相比，在使用PLPS價值函數的組合和IPLPS價值函數的組合時我們觀察到一個數量級的改進。

Acknowledgments

I sincerely thank Professor Jimmy Lee as my supervisor. Jimmy is very enthusiastic about his teaching and research, and I greatly benefit from his lessons. He teaches me a lot and having discussions with him always help develop new ideas and broaden my horizons in different aspects. I am very grateful for his invaluable advices and continuous support for my research.

I would like to thank Professor Javier Larrosa, Professor Wong Tien Tsin, and Professor Lee Pak Ching to be my examiners. Their give precious comments on the improvements of my thesis.

I also thank my members of our research groups. They contribute lots of ideas and fun for my research work. I would like to thank Terrence Mak, Andy Wu, Lee JingYing, Charles Siu and May Woo. I often benefit from their sharing of research experiences.

Last but not least, I would like to give my best wishes to my family members for their support throughout my master program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis reports work on how approximated consistency enforcement on global cost functions in weighted constraint satisfaction can be performed efficiently and effectively using linear programming techniques. We first introduce the notions of *polynomially linear projection-safe (PLPS) cost functions*, which can be modeled as *integer linear programs* with polynomial sizes. While standard consistencies can be enforced on PLPS cost functions using integer programming, computing the linear relaxation of PLPS cost functions provides a good approximation to the standard consistencies. We show further that enforcing approximated consistencies on conjunctions of *integral polynomially linear projection-safe (IPLPS)* cost functions, a special subclass of PLPS functions, is stronger than enforcing standard consistencies on the individual cost functions alone. Empirical results confirm the theoretical characterization and exhibit orders of magnitude improvements on both runtime and search space reduction. In this chapter, we first describe the Weighted Constraint Satisfaction framework before giving the motivation and goals of this thesis. We end the chapter with an overview of the structure of the rest of the thesis.

1.1 Weighted Constraint Satisfaction Problems

Weighted Constraint Satisfaction Problems (WCSPs) [45] is a soft constraint framework for modeling over-constrained problems and those with preferences. It provides a general model for different applications, such as *resource allocation* [10], *combinatorial auctions*, *electronic markets* [44], *bioinformatics* [43], *probabilistic reasoning* [37], *scheduling*, and etc.

A WCSP consists of a finite set of variables, a finite domain of possible values for each variable and a conjunction of cost functions. Each variable assignment is associated with a cost. A cost function returns a cost for each tuple. The costs could be used to represent preferences to the variable assignments.

Solving a WCSP is to find an assignment to the variables with the minimum cost. Such an assignment often represents the most preferred or the least violated situation. The basic solution technique for WCSPs is branch-and-bound search augmented with various forms of consistencies, such as NC* [24], AC* [24], FDAC* [25], and EDAC* [17]. These consistency techniques retrieve hidden information from cost functions by transporting costs and remove infeasible values from variable domains to prune the search space.

1.2 Motivation and Goal

A good library of global cost functions is essential for us to model complex real-life problems in WCSPs. A global cost function often has high arities but a special semantics. The structure of the special semantics allows special and efficient algorithms to be designed to enforce consistencies. The key concern with implementing global cost functions is tractability. Lee and Leung [30, 28] suggest three requirements for a global cost functions to be practical. First, computation of the minimum cost must be efficient. Second, projections and extensions on the cost functions can be performed efficiently. Third, projections and extensions on the

cost functions will not destroy the last two efficiency requirements. This is called *projection safety* [30, 28]. Lee and Leung further demonstrate that flow-based [48] global cost functions satisfy the first two requirements and give instances that are flow-based projection-safe. In addition, Lee *et al.* [31] show another class of cost functions, called *polynomially decomposable* cost functions, can satisfy these three requirements and give instances of cost functions which are polynomially decomposable.

Our goal is to introduce more practical global cost functions into the existing catalog. Many global cost functions are useful, yet either their minimum cost computations are NP-hard or no polynomial time algorithms are discovered yet. An example is the soft variants of the DISJUNCTIVE constraint, which schedule jobs without overlapping in a non-preemptive scheduling problem. Known algorithms for computing their minimum cost are exponential.

We first discover that the efficient minimum cost computations of global cost functions depend on the efficient enforcement of *generalized arc consistency (GAC)* of their related hard constraints. There are previous results on the NP-hardness of enforcing GAC on several global constraints, which immediately lead to the same results for the minimum cost computation of their soft variants. It is natural to ask whether there are methods to use such cost functions efficiently in different ways in WCSPs. We address this problem for the cost functions which can be modeled as integer linear programs with relaxed consistencies. By solving the integer linear programs with linear relaxation, approximations of their minimum costs are obtained and used in the enforcement of the relaxed consistencies. Such consistencies can be enforced efficiently since linear programming algorithms exhibit excellent average case behavior. We call this class of cost functions *polynomially linear projection-safe (PLPS) cost functions*.

We also consider the conjunctions of PLPS cost functions since the integer linear programming formulations of PLPS cost functions allow them to be conjoined easily. We present empirical results to demonstrate the benefits of propagating on

conjunctions in terms of both runtime and pruning in general.

We introduce and give sufficient conditions for a special subclass of PLPS cost functions, namely *integral polynomially linear projection-safe (IPLPS)* cost functions. Our results show that propagating on individual IPLPS cost functions using the standard (or relaxed since they are the same) consistencies is weaker than propagating on the conjunction of all these IPLPS cost functions using the relaxed versions of the consistencies. These results give exact characterization on the strength of the relaxed and standard consistencies on conjunctions of IPLPS cost functions as compared against the corresponding standard consistencies on individual IPLPS cost functions.

This thesis is an extension of the work by Lee and Shum [32].

1.3 Outline of the Thesis

The outline of the thesis is as the follows. Chapter 2 describes the previous work on soft constraint framework, especially the WCSP framework. We also include information about the integer linear programs, as well as the global cost functions in weighted constraint satisfaction.

Chapter 3 provides backgrounds of WCSPs with the local consistencies and global cost functions, as well as the integer linear programs. We also define the notations that we use throughout the thesis.

Chapter 4 defines *polynomially linear projection-safe (PLPS) cost functions* and relaxed consistencies for some non-tractable cost functions to be used efficiently in WCSPs. We propose a special class of global cost functions which can be modeled as integer linear programs, and call them *linear cost functions*. We give sufficient conditions to assure that a linear cost function is a PLPS cost function. We propose *relaxed consistencies*, which allow a less pruning but much more efficient (approximated) consistency enforcement. We also demonstrate the benefits of propagating on conjunctions of PLPS cost functions in terms of runtime. We give examples of

several useful PLPS cost functions and conduct experiments on them to show the efficiency of our proposed framework.

Chapter 5 defines *integral polynomially linear projection-safe (IPLPS) cost functions* as a special subclass of PLPS cost functions. We introduce and give sufficient conditions for (IPLPS) cost functions. Our results show that propagating on individual IPLPS cost functions using the standard consistencies is weaker than propagating on the conjunction of all these IPLPS cost functions using the relaxed versions of the consistencies. The results are useful when we have cost functions whose minimum cost computation is polynomial time but that for conjunctions of such cost functions is NP-hard. We show that an important class of *flow-based projection safe* [28, 30] and *polynomially decomposable* [31] cost functions belong such IPLPS cost functions. We conduct experiments to demonstrate the improvements in terms of runtime and search space of using the conjunction of IPLPS cost functions against the flow-based and polynomially decomposable approaches, as well as pure integer programming. In addition, The empirical results agree with our theoretical results.

We conclude the thesis in Chapter 6. We summarize our work on the thesis, and give future possible directions for further work.

Chapter 2

Related Work

In this chapter, we present the research areas that are related to our work. We describe various ways of handling optimization problems, including the soft constraint frameworks and integer linear programming. Next, we present an overview of some related techniques used in the WCSP framework, including the global cost functions and the local consistencies.

2.1 Soft Constraint Frameworks

In classical *constraint satisfaction problems* (CSPs) [33], all the constraints are hard constraints which can either be satisfied or violated. In many real life problems, the requirements involve preferences which is sometimes difficult to be modeled as a classic CSP. Different soft constraint frameworks are therefore proposed to solve over-constrained and optimization problems, including the probabilistic CSPs [18], fuzzy CSPs [46], and partial CSPs [19]. Here we give two examples which are closely related to our work, including the *constraint optimization problems* (COPs) [39] and *weighted constraint satisfaction problems* (WCSPs) [45].

COPs are CSPs with objective of measuring the preferences or violations. The optimality of the solutions is modeled by different objective functions based on different cost valuation structures. A way to handle COPs is the *soft-as-hard* (SasH) [39] approach. SasH models soft constraints as hard constraints, where the cost returned

by each constraint are modeled as a variable. In this model, the COPs can be solved in the same ways as classical CSPs.

Another way to model optimization problems is to model them as WCSPs, which generalizes the classical CSP framework. In a WCSP, each constraint is represented as a cost function. Instead of either be satisfied or violated, a cost function returns a cost representing the preference or violation degree. Solutions of a WCSP are the tuples with the minimum cost as the most preferred or the least violated situation.

To solve WCSPs efficiently, many consistency techniques have been proposed. Star node consistency (NC*) and star arc consistency (AC*) were developed by Larrosa and Schiex [24]. Consistency notions with stronger pruning power are developed later, including the full star directional arc consistency (FDAC*) [25] and star existential directional arc consistency (EDAC*) [17]. There are other forms of consistency notions with different pruning power appeared later, including $\varnothing$-IC [50], strong $\varnothing$-IC [28, 30], bound arc consistency (BAC) [50], virtual arc consistency (VAC) [12, 13], and *k*-consistency [11]. The use of AC*, FDAC* and EDAC* are limited to binary cost functions. They are generalized to handle high arity cost functions like global cost functions. Sanchez *et al.* [43] extended AC*, FDAC* and EDAC* for ternary cost functions. On the other hand, Cooper and Schiex [43] defined the generalized version of AC* as GAC*. The generalized version of FDAC*, called FDGAC*, is defined by Lee and Leung [28, 30], and they also show that naively generalizing the EDAC* enforcement algorithm will lead to oscillation problem when it is enforced on cost functions sharing more than one variable. They proposed a weaker form of EDGAC* with cost providing partitions called weak EDGAC* [29, 30].

There is another local consistency in WCSPs which also utilizes linear programming techniques called optimal soft arc consistency (OSAC) [14, 13]. They model the projection opportunities of table cost functions into an integer linear program. By minimizing the lower bound with linear relaxation, the maximum lower bound

can be inferred by projections is approximated.

2.2 Integer Linear Programming

Apart from soft constraint frameworks, many optimization problems can also be modeled by *integer linear programming* as *integer linear programs*. Integer linear programs are special cases of *linear programs* to represent discrete choices as integrality requirements on the variables [49], which requires the variables to take integral values. Linear programs model optimization problems with linear inequalities on continuous variables. Each linear program has linear objective function [16] which should be minimized (or maximized) such that the most preferred or the least violated situation can be obtained.

Integer linear programs can be solved by *branch-and-bound* search with a search tree, where a variable is partially fixed in each search node. At each node, the subproblem is solved by *linear relaxation* which is solved as a linear program, and the descending nodes are branched by the fractional solution of the variable to be fixed in that node until a suboptimal solution is found. The search can be speed up by different techniques like using different *branching strategies* [4], the *cutting planes* [35], and the *primal heuristics* [5].

2.3 Global Cost Functions in WCSP

A *global constraint* is a hard constraint which could be understood as *an expressive and concise condition involving a non-fixed number of variables* [2]. Since global constraints usually have have special semantics and high-arities, having efficient consistency enforcement algorithms are important for them to be used in CSPs. Global constraints are one of the keys for the success of constraint programming. Many global constraints have been proposed and studied, and a famous example is the ALLDIFF constraint [27] which is satisfied if all the variables are taking different

values. Many real life problems can be modeled by different global constraints.

Global cost functions are soft variants of global constraints with *violation measures*. Instead of either be satisfied or violated, a global cost function returns 0 if it is not violated; otherwise its violation measure is used to reflect how much the related global constraint is violated. For example, the global cost function SOFT_ALLDIFF^{dec} with the *decomposition-based* violation measure is a soft variant of the ALLDIFF constraint and returns the number of variable pairs not taking different values as its violation cost.

Different techniques are developed for some global cost functions such that they can be used in WCSP efficiently. Following the idea of Petit *et al.* [40] who use flow theories to compute the minimum cost returned by soft SOFT_ALLDIFF^{dec} , Van Hoeve *et al.* [48] develop a similar idea for the soft variants of the ALLDIFF, GCC, SAME, and REGULAR constraints.

In addition to the minimum cost computation, Lee and Leung [30, 28] further define $\mathcal{T}$ *projection-safety* for efficient use of global cost functions in WCSP. $\mathcal{T}$ *projection-safety* ensures that the property $\mathcal{T}$ is not affected by projections and extensions. If the property $\mathcal{T}$ allows the minimum cost to be computed efficiently, $\mathcal{T}$ *projection-safety* ensures that the efficient minimum cost computation is also not affected by projections and extensions. So, $\mathcal{T}$ *projection-safe* cost functions can be used in WCSPs efficiently as the consistency techniques can always be enforced efficiently. He show that some flow-based cost functions, including the SOFT_ALLDIFF^{var}, SOFT_ALLDIFF^{dec}, SOFT_GCC^{var}, SOFT_GCC^{val}, SOFT_SAME^{var} SOFT_REGULAR^{var}, and SOFT_REGULAR^{edit}, belong to flow-based projection-safe cost functions.

Other $\mathcal{T}$ *projection-safe* cost functions with different property are also discovered. Lee *et al.* [31] show that a group of cost functions called *polynomially decomposable cost functions,* including the SOFT_AMONG^{var}, SOFT_REGULAR^{var}, SOFT_REGULAR^{edit}, SOFT_GRAMMAR^{var}, MAX_WEIGHT, and MIN_WEIGHT, can

be represented as dynamic programs, which allow the minimum costs to be computed efficiently using divide-and-conquer and memorization.

Chapter 3

Background

In this chapter, we give the basic background for the rest of this thesis, including the concept of the weighted constraint satisfaction problems (WCSPs), the global cost functions, local consistencies used in WCSPs, and integer linear programming. WCSP is a framework extending CSPs to solve combinatorial problems which involve cost functions. Global cost functions are complex cost functions used to describe special structures commonly seen in most problems. Local consistencies are incorporated for efficient solving of WCSPs. Integer linear programming is a subarea in the operational research for modeling combinatorial optimization problems.

3.1 Weighted Constraint Satisfaction Problems

A *weighted constraint satisfaction problem* (WCSP) [45] is a tuple ($\mathcal{X}$, $\mathcal{D}$, $\mathcal{C}$, k). $\mathcal{X}$ 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. Each variable can only be assigned with one value in its corresponding domain. An assignment on a set of variables can be represented by a tuple ℓ. We denote $\ell[x_i]$ the value assigned to x_i, $\ell[S]$ the tuple formed from the assignment on variables in the set $S \subseteq \mathcal{X}$, and $\mathcal{L}(S)$ is a set of tuples corresponding to all possible assignments on the set of variables S. $\mathcal{C}$ is a set of *cost functions* W_S, each with scopes S. W_S maps tuples $\mathcal{L}(S)$ to a cost valuation structure $V(k) = ([0 \ldots k], \oplus, \leq)$. The structure $V(k)$ contains a set

of integers $[0, \ldots, k]$ with standard integer ordering $\leq$. Addition $\oplus$ is defined by $a \oplus b = \min(k, a + b)$. The subtraction $a \ominus b$ for $a, b \in [0 \ldots k]$ and $a \geq b$ is defined as

$$a \ominus b = \begin{cases} a - b & \text{if} \quad a \neq k \\ k & \text{otherwise} \end{cases}$$

Without loss of generality, we assume $\mathcal{C} = \{W_\varnothing\} \cup \{W_i \,|\, x_i \in \mathcal{X}\} \cup \mathcal{C}^+$. $W_\varnothing$ is the constant nullary cost function, representing the lower bound of the WCSP. W_i is a unary cost function associated with variable $x_i \in \mathcal{X}$. We may also call the costs of the unary cost functions associated with each value of the variables as the *unary cost* of that value. C^+ is a set of cost functions with scopes of two or more variables. If a cost function has a scope of only two variables $\{x_i, x_j\}$, we call it a *binary cost function* and we use W_{ij} to denote it.

Example 3.1. *Figure 3.1 shows a WCSP with two variables $\mathcal{X} = \{x_1, x_2\}$ with domains $D(x_1) = \{a, b, c\}$ and $D(x_2) = \{a, b\}$ respectively, and three cost functions W_1, W_2 and $\mathcal{C}^+ = \{W_{12}\}$ given as tables. W_1 and W_2 are unary cost functions, and W_{12} is a binary cost function. The lower bound $W_\varnothing$ equals to 0 and the upper bound k is set to be 5.*

x_1	W_1
a	1
b	0
c	5

(a) W_1

x_2	W_2
a	1
b	2

(b) W_2

x_1	x_2	W_{12}
a	a	2
a	b	1
b	a	0
b	b	0
c	a	0
c	b	0

(c) W_{12}

Figure 3.1: *A WCSP with two variables and three cost functions*

The graphical representation of this WCSP is shown in figure 3.2. A rectangle represents a variable domain, where each value is represented by a circle inside the rectangle of that variable. The numbers in the circles stand for the unary costs

given by the unary cost functions, which is omitted if the corresponding value has zero unary cost. An edge between two circles represents the binary cost associated to the tuple formed by the two values represented by the two circles. A label ω is associated on each edge representing the binary cost of the associated tuple. The label ω is omitted if $\omega = 1$. The edge is omitted if $\omega = 0$.

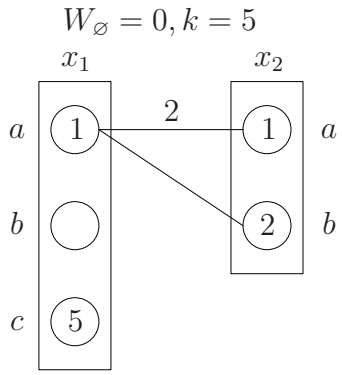


Figure 3.2: *Graphical representation of a WCSP*

The cost of a tuple ℓ for a WCSP corresponding to an assignment on $\mathcal{X}$ is defined as

$$cost(\ell) = W_\varnothing \oplus \bigoplus_{x_i \in \mathcal{X}} W_i(\ell[x_i]) \oplus \bigoplus_{W_S \in \mathcal{C}^+} W_S(\ell[S])$$

A tuple ℓ is *feasible* if $cost(\ell) < k$. Our goal is to find a tuple ℓ which has the minimum cost among all the feasible tuples, and such a tuple is a *solution* of the WCSP. For convenience, we write $\min\{W_S\}$ to denote $\min\{W_S(\ell) \mid \ell \in \mathcal{L}(S)\}$.

Example 3.2. *Given the WCSP shown in Example 3.1. The cost of each tuple is shown as follows.*

$$cost(a,a) = 4 \quad cost(b,a) = 1 \quad cost(c,a) = 6$$
$$cost(a,b) = 4 \quad cost(b,b) = 2 \quad cost(c,b) = 7$$

The tuples (c,a) and (c,b) are not feasible since their costs are equal to or greater than the upper bound $k = 5$. Besides, among all tuples, (b,a) has the minimum cost and thus it is the solution of this WCSP.

3.1.1 Branch and Bound Search

Solutions of a WCSP can be found by systematic search. A systematic search method guarantees to find a solution of a WCSP if there exists one, or prove no solution. A type of systematic search techniques commonly used for WCSPs is the *branch and bound* (BnB) search algorithm. It traverses the search tree of all possible assignments in a depth-first left-to-right manner. Given a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$, the procedure `BranchandBound(`$\mathcal{X}, \mathcal{D}, \mathcal{C}, 0, k, \varnothing$`)` in Algorithm 1 returns one of its solutions if there exists at least one, or proves no solution by returning k [26].

```
1  Procedure BranchAndBound(X, D, C, W_∅, k, l) begin
2      if X = ∅ then
3          store(l);
4          return W_∅;
5      x_i ← chooseVar(X);
6      foreach v ∈ D(x_i) do
7          l' ← l ∪ {x_i ↦ v};
8          W'_∅ ← W_∅ ⊕ W_i(v);
9          C' ← lookAhead(C, {x_i ↦ v});
10         enforceLocalConsistency*();
11         if W_∅ ≥ k then return k;
12         k ← BranchAndBound(X\{x_i}, D, C, W'_∅, k, l');
13     return k;
14 Procedure lookAhead(C, {x_i ↦ v}) begin
15     C' ← C\{W_i};
16     foreach W_ij ∈ C do
17         foreach b ∈ D(x_j) do
18             W'_j(b) ← W'_j(b) ⊕ W_ij(v, b);
19         C' ← C'\{W_ij};
20     return C'
```

Algorithm 1: Branch and Bound Search Algorithm for a WCSP

During the search, a *currently best feasible tuple* is kept as the upper bound. Initially, the upper bound is set to be k, and updated when a better feasible tuple is found. On each search node, a value is assigned to x_i and the WCSP is reduced

to a new WCSP ($\mathcal{X}\backslash\{x_i\}, \mathcal{D}, \mathcal{C}, k$). $\mathcal{C}'$ is formed by the procedure `lookAhead()`, which reduces the cost functions involving x_i by removing x_i from that cost function.

The procedure `enforceLocalConsistency*()` enforces the local consistency on the current WCSP, which will be discussed in the next section and we omit the details for the moment.

The lower bound at this node $W_\varnothing$ is then evaluated. If it is not less than the upper bound, it proves that no feasible tuple with a cost lower than that of the currently best feasible tuple can appear in the search tree beneath this search node. In this case the algorithm immediately backtracks.

If $\mathcal{X}$ is reduced to an empty set, all variables are assigned, and the lower bound $W_\varnothing$ equals to the cost of the corresponding tuple. If such a tuple is found, this tuple is stored as a currently best feasible tuple and the upper bound k is updated to the cost of this tuple, such that the algorithm has to find a new tuple having the cost lower than that of the currently best feasible tuple. Finally the algorithm returns the best feasible tuple found as a solution of the WCSP.

This algorithm can also be applied on non-binary cost functions by modifying the procedure `lookAhead()`. Figure 3.3 shows a search tree for solving the WCSP in Example 3.1 using the branch and bound search algorithm.

3.1.2 Local consistencies in WCSP

Different local consistency techniques can be incorporated with the basic branch and bound search with the procedure `enforceLocalConsistency*()`. They are capable of removing infeasible values in the domains and deducing a lower bound of the minimum cost, where the lower bound can be used to trigger the backtrack from the search nodes. The consistency notions for WCSPs are achieved by equivalence preserving transformation.

Definition 3.3. *Given two WCSPs* $P_1 = (\mathcal{X}, \mathcal{D}_1, \mathcal{C}_1, k)$ *and* $P_2 = (\mathcal{X}, \mathcal{D}_2, \mathcal{C}_2, k)$*,* P_1

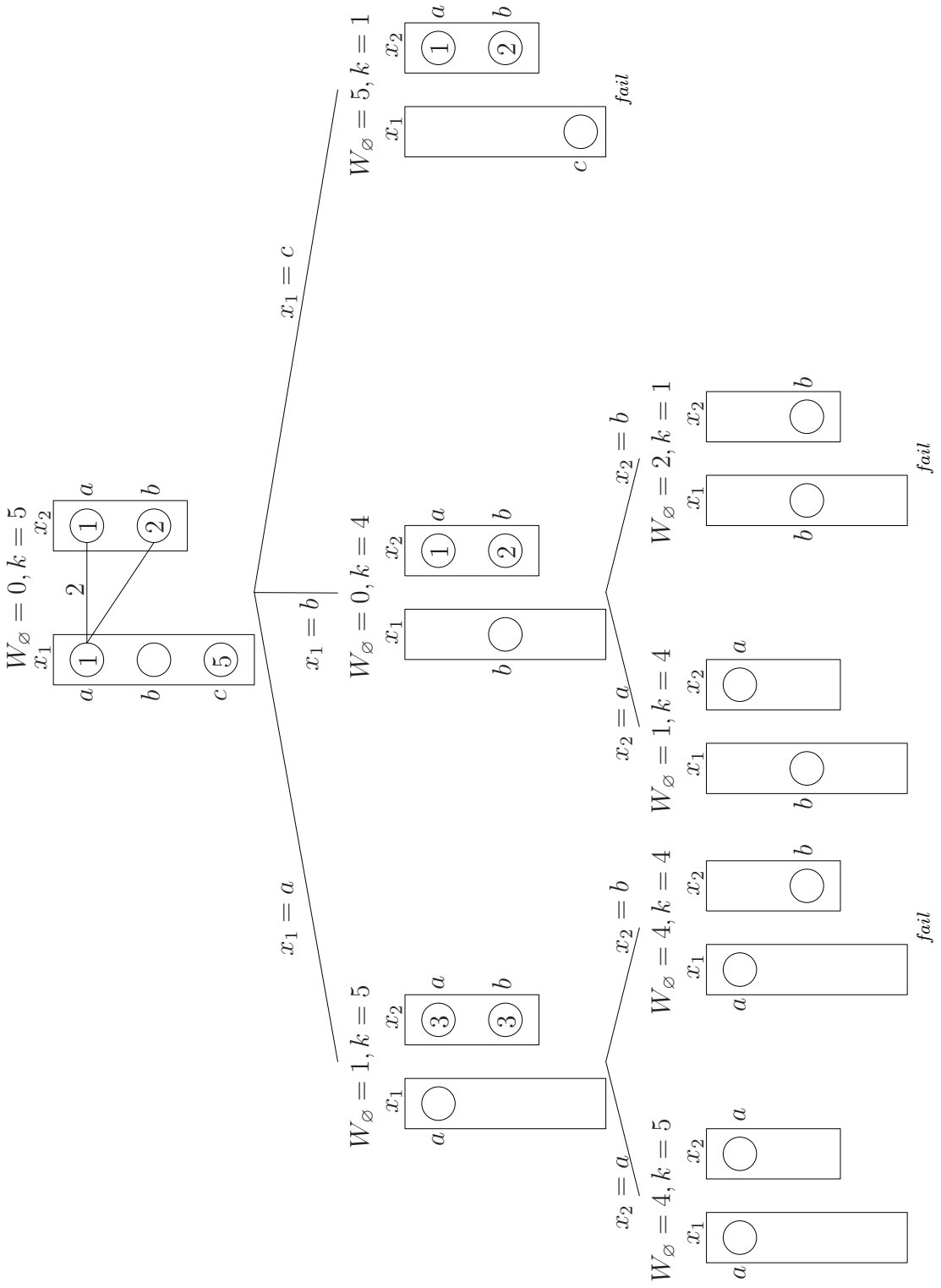


Figure 3.3: A branch and bound search to solve a WCSP

is equivalent to P_2 iff for all feasible tuples $\ell \in \mathcal{L}(\mathcal{X})$ in both problems, $cost_{P_1}(\ell) = cost_{P_2}(\ell)$.

In the following, we briefly discuss those consistency notions in WCSPs including NC* [24], (G)AC* [24, 15], FD(G)AC* [25, 30], and (weak) ED(G)AC* [17, 30].

The enforcement of those consistencies involves finding the minimum costs of the cost functions, and moving those costs between cost functions by *projections* and *extensions* [11]. Projections move costs from n-nary cost functions to unary cost functions and from unary cost functions to the nullary one $W_\varnothing$. Given $S_2 \subset S_1$, a projection of cost α from W_{S_1} to W_{S_2} with respect to $\ell \in \mathcal{L}(S_2)$ is a transformation of (W_{S_1}, W_{S_2}) to (W'_{S_1}, W'_{S_2}), where

$$W'_{S_1}(\ell') = \begin{cases} W_{S_1}(\ell') \ominus \alpha & \text{if } \ell'[S_2] = \ell \\ W_{S_1}(\ell') & \text{otherwise} \end{cases}$$
$$W'_{S_2}(\ell') = \begin{cases} W_{S_2}(\ell') \oplus \alpha & \text{if } \ell' = \ell \\ W_{S_2}(\ell') & \text{otherwise} \end{cases}$$

If $S_2 = \varnothing$, it is a projection to $W_\varnothing$. Extensions are the inverse of projections, and are defined similarly. We assume that the minimum cost of the cost functions $\min\{W'_S\}$ cannot be smaller than 0 after a projection or extension operation.

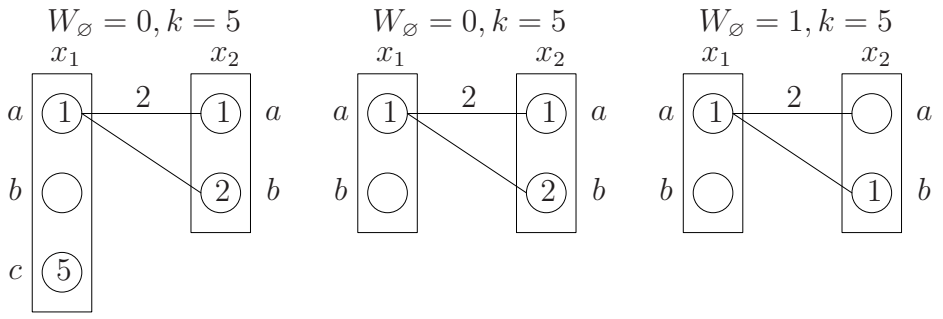
Star Node Consistency

Definition 3.4. *[24] Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$.*

- *A value $v \in D(x_i)$ where $x_i \in \mathcal{X}$ is* star node consistent *(NC*) if $W_\varnothing \oplus W_i(v) < k$.*
- *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 $W_i(v) = 0$. Such a value is called a* unary support *of x_i.*
- *P is NC* if all its variables are NC*.*

NC* increases $W_\varnothing$ from unary cost functions and remove infeasible values. It helps the branch and bound search to detect unsatisfiability by checking if empty domain exists or $W_\varnothing$ reaches the upper bound k. We use the WCSP from Figure 3.2 as an example.

Example 3.5. *The WCSP in Figure 3.4(a) is not NC*. The value $c \in D(x_1)$ is not NC* since $W_1(c) = 5 = k$. Besides, x_2 is not NC* since no value have zero unary cost in $D(x_2)$. To transform the WCSP into an equivalent WCSP which is NC*, we remove the value c from the domain of $D(x_1)$ as shown in Figure 3.4(b). After that we project a cost of 1 from W_2 to $W_\varnothing$ as shown in Figure 3.4(c) and the resultant WCSP is NC*.*



(a) The original WCSP (b) After $c \in D(x_i)$ is removed (c) After projecting from W_2 to $W_\varnothing$

Figure 3.4: *Enforcing NC* on a WCSP*

The procedure `enforceNC*()` in Algorithm 2 enforce NC* for a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$ [26]. The algorithm first projects cost from each variable. Then it removes infeasible values according to the lower bound $W_\varnothing$. The function `unaryProject()` projects a suitable cost from W_i to $W_\varnothing$ to produce a unary support, while the function `pruneVal()` removes the infeasible values which is not NC*.

(Generalized) Star Arc Consistency

Definition 3.6. *[24] Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$.*

```
1  Procedure enforceNC*()begin
2      foreach x_i ∈ 𝒳 do
3          unaryProject(x_i);
4      foreach x_i ∈ 𝒳 do
5          pruneVal(x_i);
6  Function unaryProject(x_i)begin
7      α := k;
8      foreach v ∈ D(x_i) do
9          if α > W_i(v) then α := W_i(v);
10     W_∅ := W_∅ ⊕ α;
11     foreach v ∈ D(x_i) do
12         W_i(v) := W_i(v) ⊖ α;
13 Function pruneVal(x_i):Boolean begin
14     flag := false;
15     foreach v ∈ D(x_i) s.t. W_i(v) ⊕ W_∅ = k do
16         D(x_i) := D(x_i)\{v};
17         flag := true;
18     return flag;
```

Algorithm 2: Enforcing NC* for a WCSP

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

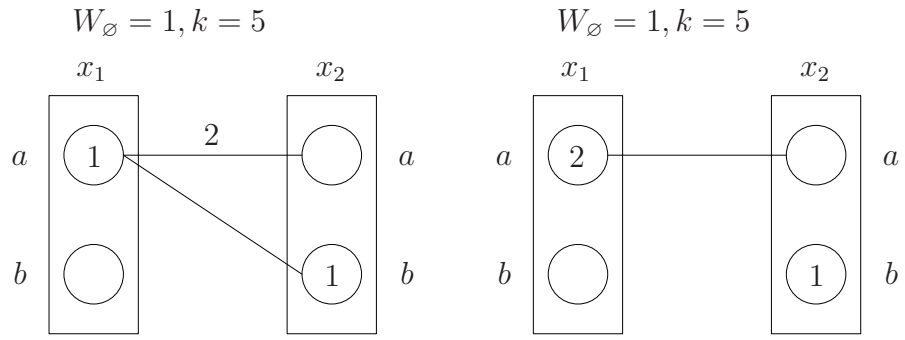
- *A variable $x_i \in \mathcal{X}$ is AC* if it is NC* and each value in $D(x_i)$ is AC* with respect to every binary cost function over x_i*

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

AC* helps extract cost information hidden in binary cost functions and expresses it as unary costs. We use the WCSP from Figure 3.4(c) as an example.

Example 3.7. *The WCSP in Figure 3.5(a) is NC* but not AC*. The value $a \in D(x_1)$ is not AC*. If a is assigned to x_1, the binary cost function W_{12} returns a cost*

of at least one no matter what value x_2 takes. As shown in Figure 3.5, we can transform the WCSP into an equivalent one which is AC by projecting a cost of 1 from $W_{12}(x_1 = a)$ to $W_1(a)$.*



(a) The original WCSP

(b) After projecting from $W_{12}(x_1 = a)$ to $W_1(a)$

Figure 3.5: *Enforcing AC* on a WCSP*

AC* can only be enforced on binary cost functions, but it can be generalized to *generalized star arc consistency* (GAC*) [15] in order to be enforced on n-ary cost functions.

Definition 3.8. *A variable $x_i \in S$ is* GAC* *[15] with respect to a cost function W_S if:*

- *x_i is NC*, and;*
- *for each value $v_i \in D(x_i)$, there exists values $v_j \in D(x_j)$ for all $j \neq i$ and $x_j \in S$ so that they form a tuple ℓ with $W_S(\ell) = 0$. ℓ is a* simple support *of v_i with respect to W_S.*

A WCSP is GAC* *iff all variables are GAC* with respect to all cost functions in $\mathcal{C}$.*

The second requirement can be reformulated as:

For each value $v_i \in D(x_i)$, $\min\{W_S(\ell) \mid \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} = 0$.

Lee and Leung [30, 28] gives the algorithm for enforcing GAC*. The procedure `enforceGAC*()` in Algorithm 3 enforce GAC* for a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$. The propagation queue $\mathcal{Q}$ stores a set of variables x_j. If $x_j \in \mathcal{Q}$, all variables in the cost functions involving x_j are potentially not GAC*. Initially all variables are in $\mathcal{Q}$. A variable x_j is pushed into $\mathcal{Q}$ after values are removed from $D(x_j)$. At each iteration, an arbitrary variable x_j is removed from the queue by the function `pop()` at line 4. The existence of a simple support with respect to the non-unary cost function C_S for the value in $D(x_i)$, where $x_i \in S$, is enforced by the function `findSupport()` at line 8. Lastly, the infeasible values are removed by the function `pruneVal()` at lines 9 and 12. If a value from $D(x_i)$ is removed, the simple supports of other variables may be destroyed and x_i is pushed into $\mathcal{Q}$. Lee and Leung [30, 28] also proves that this algorithm must terminal by stating its complexity.

Full Star Directional (Generalized) Arc Consistency

Definition 3.9. *[25] Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$.*

- *The value $b \in D(x_j)$ is a* full support *of a value $a \in D(x_i)$ if $W_{ij}(a, b) \oplus W_j(b) = 0$).*
- *The value $a \in D(x_i)$ is* directional arc consistent *with respect to a binary constraint W_{ij} where $j > i$ if there exists a full support in $D(x_j)$.*
- *A variable x_i is* star directional arc consistent *(DAC*) if it is NC* and each value in its domain is directional arc consistent with repsect to all binary constraints W_{ij} where $j > i$.*
- *P is* fully star arc consistent *(FDAC*) if all variables are AC* and DAC*.*

FDAC* also helps extract hidden cost information and expresses it as unary costs. We use the WCSP from Figure 3.5(c) as an example.

Example 3.10. *The WCSP in Figure 3.6(a) is not AC* but not FDAC*. The value $a \in D(x_1)$ is not FDAC* since it cannot find a full support with respect to C_{12}. To*

```
1  Procedure enforceGAC*()begin
2      Q := X;
3      while Q ≠ ∅ do
4          x_j :=pop(Q);
5          flag := false;
6          foreach W_S s.t. {x_j} ⊂ S do
7              foreach x_i ∈ S \ {x_j} do
8                  flag := flag ∨ findSupport(W_S, x_i);
9                  if pruneVal(x_i) then Q := Q ∪ {x_i};
10         if flag then
11             foreach x_i ∈ X do
12                 if pruneVal(x_i) then Q := Q ∪ {x_i};
13 Function findSupport(W_S, x_i):Boolean begin
14     flag := false;
15     foreach v ∈ D(x_i) do
16         α := min{W_S(ℓ)|ℓ ∈ L(S) ∧ ℓ[x_i] = v};
17         if W_i(v) = 0 ∧ α > 0 then flag := true;
18         W_i(v) := W_i(v) ⊕ α;
19         foreach ℓ ∈ L(S) s.t. ℓ[x_i] = v do
20             W_S(ℓ) := W_S(ℓ) ⊖ α;
21     unaryProject(x_i);
22     return flag;
```

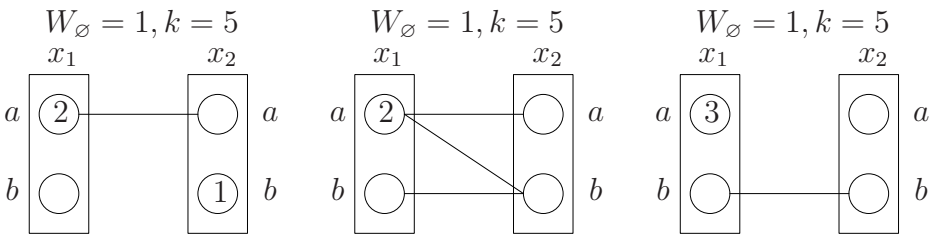
Algorithm 3: Enforcing GAC* for a WCSP

transform the WCSP into an equivalent one which is FDAC, we extend a cost of 1 from $W_2(b)$ to W_{12} as shown in Figure 3.6(b). After that we can project a cost of 1 from W_{12} to $C_1(a)$ and the resultant WCSP is FDAC* as shown in Figure 3.6(c).*

Similar to AC*, FDAC* can only be enforced on binary cost functions and it can be generalized to *full star generalized arc consistency* (FDGAC*) [30, 28] in order to be enforced on n-ary cost functions.

Definition 3.11. *A variable $x_i \in S$ is* star directional generalized arc consistent *(DGAC*) [30, 28] with respect to a cost function W_S if:*

- *x_i is NC*, and;*



(a) The original WCSP (b) After extending from $W_2(b)$ to W_{12} (c) After projecting from W_{12} to $C_1(a)$

Figure 3.6: *Enforcing FDAC* on a WCSP*

- *for each value* $v \in D(x_i)$*, there exists values* $v_j \in D(x_j)$ *for all* $j \neq i$ *and* $x_j \in S$ *such that they form a tuple* ℓ *with* $W_S(\ell) \oplus \bigoplus_{x_j|j>i} W_j(\ell[x_j]) = 0$*.* ℓ *is a* full support *of* v_i *with respect to* W_S*.*

The second requirement can be reformulated as:

> *For each value* $v_i \in D(x_i)$*,* $\min\{W_S(\ell) \oplus \bigoplus_{x_j|j>i} W_j(\ell[x_j]) \,|\, \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} = 0$*.*

A WCSP is fully star directional generalized arc consistent *(FDGAC*) iff it is GAC* and all variables are DGAC* with respect to all cost functions in* $\mathcal{C}$*.*

The procedure `enforceFDGAC*()` in Algorithm 4 enforces FDGAC* for a WCSP [30, 28]. The propagation queues $\mathcal{Q}$ and $\mathcal{R}$ store a set of variables. If $x_j \in \mathcal{Q}$, all variables involving in the same cost functions as x_j are potentially not GAC*; if $x_j \in \mathcal{R}$, the variables x_i with $j > i$ involving in the same cost functions as x_j are potentially not DGAC*. A variable x_j is pushed into $\mathcal{Q}$ only after values are removed from $D(x_j)$, or the unary support of x_j is modified. At each iteration, GAC* is enforced first by the first inner while-loop from line 4 to 14. DGAC* is then enforced by the second inner while-loop from lines 15 to 20. Enforcing DGAC* follows the ordering from the largest index to the smallest index such that the full supports of values in the do- mains of variables with smaller indices are not destroyed by DGAC*-enforcement for those with larger indices. The variable with the largest index in $\mathcal{R}$ is removed from $\mathcal{R}$ by the function

`popMax()` in constant time. DGAC* enforcement is performed by the procedure `findFullSupport()`. Lastly, NC* is enforced by the for-loop from lines 21 to 23. Lee and Leung [30, 28] also proves that this algorithm must terminal by stating its complexity.

(Weak) Star Existential Directional (Generalized) Arc Consistency

Definition 3.12. *[17] Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$.*

- *A variable x_i is* star existential arc consistent *(EAC*) if there exists at least one value $v \in D(x_i)$ such that $W_i(v) = 0$ and it has a full support with respect to every binary cost function W_{ij}. Such a value v is called the* fully supported value *of x_i.*
- *P is* existential arc consistent *(EAC*) if all variables are NC* and EAC*.*
- *P is* star existential directional arc consistent *(EDAC*) if it is FDAC* and EAC*.*

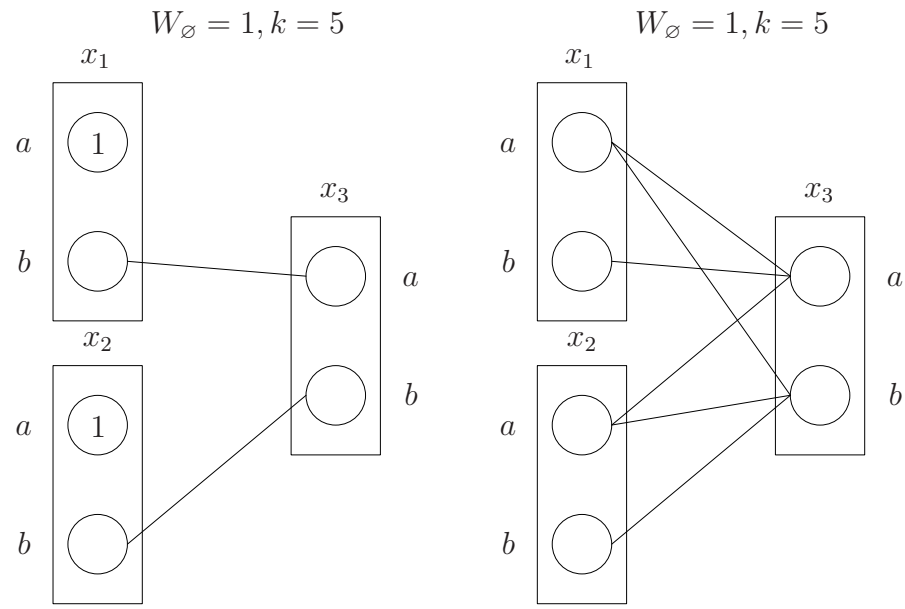
By enforcing EDAC*, $W_\varnothing$ can be increased further. We use the following example to demonstrate this idea.

Example 3.13. *The WCSP shown in Figure 3.7(a) is FDAC* but not EAC*. Consider the variable x_3, both values a and b must take a cost of at least 1, since $W_{23}(v, a) \oplus W_2(v) \geq 1$ for every $v \in D(x_2)$ and $W_{13}(v, b) \oplus W_1(v) \geq 1$ for every $v \in D(x_1)$. As a result, the solution should have a cost of at least 1. To further increase $W_\varnothing$, we first extend a cost of 1 from $W_1(b)$ to W_{13} and also a cost of 1 from $W_2(a)$ to W_{23} as shown in Figure 3.7(b). Then we project a cost of 1 from W_{13} to $W_3(b)$ and another cost of 1 from W_{23} to $W_3(a)$ as shown in Figure 3.7(c). Finally we enforce NC* on x_3 and the lower bound $W_\varnothing$ is increased by 1, and the resultant WCSP is EDAC* as shown in Figure 3.7(d).*

Lee and Leung [30, 29] showed that a naive generalization of EDAC* to high arity cost functions is not always enforceable, i.e. the algorithm may not terminate.

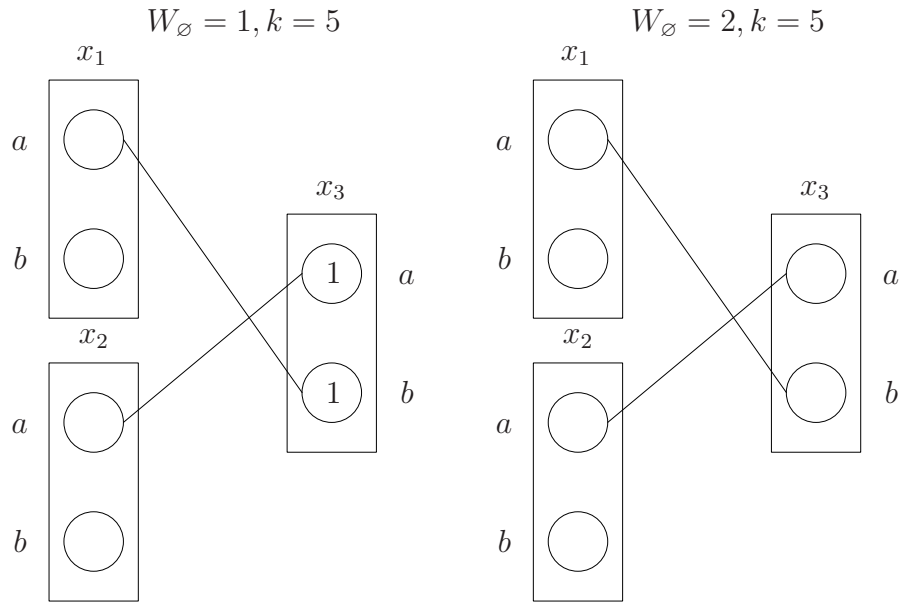
```
1  Procedure enforceFDGAC*()begin
2      R := Q := X;
3      while R ≠ ∅ ∨ Q ≠ ∅ do
4          while Q ≠ ∅ do
5              x_j :=pop(Q);
6              flag := false;
7              foreach W_S s.t. {x_j} ⊂ S do
8                  foreach x_i ∈ S \ {x_j} do
9                      R := R ∪ {x_i};
10                     flag := true;
11             if flag then
12                 foreach x_i ∈ X s.t. pruneVal(x_i) do
13                     Q := Q ∪ {x_i};
14                     R := R ∪ {x_i};
15         while R ≠ ∅ do
16             x_j := popMax(R);
17             foreach W_S s.t. {x_j} ⊂ S do
18                 for i = n downto 1 s.t. x_i ∈ S\{x_j} do
19                     if findFullSupport(W_S, x_i, {x_u|u > i} ∩ S)
                       then
20                         R := R ∪ {x_i};
21         foreach x_i ∈ X s.t. pruneVal(x_i) do
22             Q := Q ∪ {x_i};
23             R := R ∪ {x_i};
24 Function findFullSupport(W_S, x_i, U):Boolean begin
25     foreach x_j ∈ U do
26         foreach v ∈ D(x_j) do
27             foreach ℓ ∈ L(S) s.t. ℓ[x_j] = v do
28                 W_S(ℓ) := W_S(ℓ) ⊕ W_j(v_j);
29             W_j(v_j) := 0;
30     flag := findSupport(W_S, x_i);
31     foreach x_j ∈ U do findSupport(W_S, x_j);
32     unaryProject(x_i);
33     return flag;
```

Algorithm 4: Enforcing FDGAC* for a WCSP



(a) The original WCSP

(b) After extensions



(c) After projections

(d) After enforcing NC* on x_3

Figure 3.7: *Enforcing EDAC* on a WCSP*

They define a weaker form of EDGAC* with fully support sets called *weak star existential directional generalized arc consistency* (weak EDGAC*).

Definition 3.14. *The* fully supported set $U(W_S, x_i)$ *for a variable* x_i *and a cost function* W_S *with* $x_i \in S$ *is a set of variables such that:*

- $U(W_S, x_i) \subseteq S$*;*
- $U(W_S, x_i) \cap U(W_T, x_i) = \varnothing$ *for two different cost functions* $W_S, W_T \in \mathcal{C}$*, and;*
- $\bigcup_{W_S \in \mathcal{C} \wedge x_i \in S} U(W_S, x_i) = (\bigcup_{W_S \in \mathcal{C} \wedge x_i \in S} S) \backslash \{x_i\}$.

They give a simple way to compute the fully supported set for a variable x_i in 5.

```
1 Procedure findFullySupportedSet()begin
2     Y = (⋃ W_{S_j} ∈ C ∧ x_i ∈ S_j S_j)\{x_i};
3     foreach W_{S_j} ∈ C s.t. x_i ∈ S_j do
4         U(C_{S_j}, x_i) = Y ∩ S_j;
5         Y = Y\S_j;
```

Algorithm 5: Finding the fully supported set for a variable x_i

Definition 3.15. *Given a WCSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$ *and any fully supported set* $U(W_S, x_i)$ *for each variable* $x_i \in \mathcal{X}$ *and each cost function* $W_S \in \mathcal{C}$*. A variable* $x_i \in S$ *is* weak star existential generalized arc consistent *(weak EGAC*) [30, 29] if:*

- x_i *is NC*, and;*
- *there exists a value* $v \in D(x_i)$ *such that for each cost function* $W_S \in \mathcal{C}$ *with* $x_i \in S$ *and* $U(W_S, x_i)$*, there exists values* $v_j \in D(x_j)$ *for all* $j \neq i$ *and* $x_j \in S$ *such that they form a tuple* ℓ *with* $W_S(\ell) \oplus \bigoplus_{x_j | j \in U(W_S, x_i)} W_j(\ell[x_j]) = 0$*.* v *is a* weak fully supported value *of* x_i*.*

The second requirement can be reformulated as:

there exists a value $v \in D(x_i)$ *such that for each cost function* $W_S \in \mathcal{C}$ *with* $x_i \in S$ *and* $U(W_S, x_i)$, $\min\{\bigoplus_{x_i \in S} W_S(\ell) \oplus \bigoplus_{x_j | j \in U(W_S, x_i)} W_j(\ell[x_j]) \mid \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} = 0$.

A WCSP is weak star existential directional generalized arc consistent *(weak EDGAC*) iff it is FDGAC* and all variables are weak EDGAC*.*

The procedure `enforceWeakEDGAC*()` in Algorithm 6 enforces weak EDGAC* of a WCSP [30, 29]. The fully supported set is first computed at line 2. The procedure makes use of four propagation queues $\mathcal{P}, \mathcal{Q}, \mathcal{R}$ and $\mathcal{S}$. If $x_i \in \mathcal{P}$, the variable x_i is potentially not weak EGAC* due to a change in unary costs or a removal of values in some variables. If $x_j \in \mathcal{R}$, the variables x_i with $j > i$ involving in the same cost function as x_j are potentially not DGAC*. IF $x_j \in \mathcal{Q}$, all variables in the same cost function as x_j are potentially not GAC*. The propagation queue $\mathcal{S}$ helps build cost functions as x_j are potentially not GAC*. The propagation queue $\mathcal{S}$ helps build $\mathcal{P}$ efficiently. The procedure consists of three inner-while loops and one for-loop. The first inner-while loop from from lines 5 to 9 enforces weak EGAC* on each variable by the procedure `findExistentialSupport()` at line 7. If the procedure returns true, a projection from some constraints to C_i has been performed. The weak fully-supported values of other variables may be destroyed. Thus, the related variables are pushed back to $\mathcal{P}$ for revision at line 9. The second inner-while loop from lines 11 to 17 enforces DGAC*, while the third inner-while loop from lines 18 to 25 enforces GAC*. A change in unary cost requires re-examining DGAC* and weak EGAC*, which is done from lines 8 to 9 and from lines 16 and 17. Lastly, NC* is enforced by the for-loop from lines 26 to 29. Again, if a value in $D(x_i)$ is removed, GAC*, DGAC* or weak EGAC* may be destroyed, and x_i are pushed into the corresponding queues for re-examination. Lee and Leung [30, 29] also proves that this algorithm must terminal by stating its complexity.

```
1  Procedure enforceWeakEDGAC*()begin
2      foreach x_i ∈ 𝒳 do findFullySupportedSet(x_i)
       𝒮 := ℛ := 𝒬 := 𝒳;
3      while 𝒮 ≠ ∅ ∨ ℛ ≠ ∅ ∨ 𝒬 ≠ ∅ do
4          𝒫 := 𝒮 ∪ ⋃_{x_i∈S,W_S∈𝒞}(S\{x_i});
5          while 𝒫 ≠ ∅ do
6              x_i := pop(𝒫);
7              if findExistentialSupport(x_i) then
8                  ℛ := ℛ ∪ {x_i};
9                  𝒫 := 𝒫 ∪ ({x_j|x_i, x_j ∈ W_S, W_S ∈ 𝒞}\{x_i});
10         S := ∅;
11         while ℛ ≠ ∅ do
12             x_u := popMax(ℛ);
13             foreach W_S s.t. {x_u} ⊂ S do
14                 for i = n downto 1 s.t. x_i ∈ S\{x_u} do
15                     if
                       findFullSupport(W_S, x_i, {x_j|j > i ∧ x_j ∈ S})
                       then
16                         𝒮 := 𝒮 ∪ {x_i};
17                         ℛ := ℛ ∪ {x_i};
18         while 𝒬 ≠ ∅ do
19             x_u := pop(𝒬);
20             flag := false;
21             foreach C_S s.t. {x_u} ⊂ S do
22                 foreach x_i ∈ S\{x_u} do
23                     if findSupport(C_S, x_i) then
24                         𝒮 := 𝒮 ∪ {x_i};
25                         ℛ := ℛ ∪ {x_i};
26         foreach x_i ∈ 𝒳 s.t. pruneVal(x_i) do
27             𝒮 := 𝒮 ∪ {x_i};
28             𝒬 := 𝒬 ∪ {x_i};
29             ℛ := ℛ ∪ {x_i};
30 Function findExistentialSupport(x_i):Boolean begin
31     flag := false;
32     α := min_{a∈D(x_i)}{W_i(a) ⊕ ⨁_{x_i∈S,W_S∈𝒞} min_{ℓ[x_i]=a}{W_S(ℓ) ⊕
       ⨁_{x_j∈U(W_S,x_i)} W_j(ℓ[x_j])}};
33     if α > 0 then
34         flag := true;
35         foreach W_S ∈ 𝒞 s.t. x_i ∈ S do
36             findFullSupport(W_S, x_i, U(W_S, x_i));
37     return flag;
```

Algorithm 6: Enforcing weak EDGAC* for a WCSP

3.1.3 Global Cost Functions

The cost functions used in WCSPs can be represented as tables, where each entry specifies the cost of a tuple in each cost function. However the size of the corresponding table is exponential to the number of the variables in a cost function. Thus with such table representation, only binary and ternary cost functions were used practically in WCSPs.

In contrast, a *global cost function* is a cost function with special semantics used in the WCSP framework. Usually, there are efficient algorithms designed for the consistency enforcement.

We denote a global cost function as $\text{SOFT\_GC}^{\mu}(S)$ if it is derived from the corresponding hard *global constraint* GC(S) of the variable scope S with a *violation measure* μ, where global constraints are used in the CSP framework. $\text{SOFT\_GC}^{\mu}(S)$ returns 0 *iff* a given tuple ℓ on S satisfies GC. If ℓ violates GC, $\text{SOFT\_GC}^{\mu}(S)$ returns $\mu(\ell)$ using the violation measure to reflect how much the GC is violated. To handle global cost functions which are usually of high-arity, common consistencies are generalized to GAC* [15] and FDGAC* [30, 28], and weak EDGAC* [30, 29].

We give an example to show that by using global cost functions, more hidden information may be extracted during the consistency enforcement than using the corresponding binary cost functions. The global cost function $\text{SOFT\_ALLDIFF}^{dec}(S)$ returns 0 when variables in S take distinct values, otherwise $\text{SOFT\_ALLDIFF}^{dec}(S) = \{x_i \neq x_j | x_i, x_j \in S \wedge i \neq j\}$.

Example 3.16. *Given a WCSP with three variables $\mathcal{X} = \{x_1, x_2, x_3\}$, where $D(x_1) = D(x_2) = \{a, b\}$ and $D(x_3) = \{a, b, c\}$ with all unary costs equal to 0. There are three binary cost functions W_{12}, W_{23}, and W_{13} where a cost of 0 is taken if $x_1 \neq x_2$, $x_1 \neq x_3$ and $x_2 \neq x_3$, otherwise a cost of 1 is taken. It is AC* since every variable is AC* with respect to all related cost functions. If the cost function is replaced by a $\text{SOFT\_ALLDIFF}^{dec}(\{x_1, x_2, x_3\})$ cost function, the WCSP is not AC*, since values $a, b \in D(x_3)$ has no support with respect to the $\text{SOFT\_ALLDIFF}^{dec}$ cost function.*

Lee and Leung [30, 28] defines $\mathcal{T}$ *projection-safety*. A cost function W_S is $\mathcal{T}$ *projection-safe* if (a) W_S satisfies property $\mathcal{T}$, and (b) W'_S satisfies property $\mathcal{T}$, where W'_S is obtained from W_S by a valid sequence of projections or extensions. In other words, the property $\mathcal{T}$ is preserved on W_S under projections and extensions. Given a $\mathcal{T}$ projection-safe cost function W_S, if the property $\mathcal{T}$ allows an efficient computation of the minimum cost of W_S, it is guaranteed that the minimum cost of W_S can still be computed efficiently after projections and extensions.

Two useful properties $\mathcal{T}$ are flow-basedness and polynomially decomposable. *Flow-based projection-safe cost functions* [30, 28] can be represented as flow networks, the minimum cost of which can be computed efficiently by flow algorithms. *Polynomially decomposable cost functions* [31] can be represented as dynamic programs, which allow the minimum costs to be computed efficiently using divide-and-conquer and memorization.

3.2 Integer Linear Programming

In this thesis, we formulate global cost functions by *integer linear programs* [49]. An integer linear program I is defined as follows:

$$z = \min(c^T X)$$

$$aX \leq b$$

$$l \leq X \leq u$$

$$X \in \mathbb{Z}^n$$

X is a set of *variables* such that $X = \{x_1, x_2, \ldots, x_n\}$ and $aX \leq b$ are *linear constraints* where $a \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$ given n is the number of variables and m is the number of problem constraints. $z = \min(c^T X)$ is the *objective function* where $c \in \mathbb{Q}^n$. l and u are *lower* and *upper bounds* on the variables X where

$l \in (\mathbb{Q}^n \cup \{-\infty\})$ and $l \in (\mathbb{Q}^n \cup \{\infty\})$. Solving an integer linear program is to find values for the variables X *minimizing (or maximizing) the objective function* $z = \min(c^T X)$ while satisfying all the linear constraints $aX \leq b$.

A *linear program* [16] is a special case of an integer linear program where all the variables are not longer required to be integers. So the integrality requirement $X \in \mathbb{Z}^n$ is removed such that $X \in \mathbb{R}^n$.

An *assignment* γ represents the values taken by the variables in X. A *feasible solution* is an assignment γ that satisfies all problem constraints $aX \leq b$. An *optimal feasible solution* is an assignment γ representing a feasible solution and the objective function $c^T X$ gives the minimal value. We call the value of the objective function z from an optimal feasible solution of $z = \min c^T X$ as the *minimum* of I or $\min(I)$.

We use integer linear programs to model global cost functions in WCSPs as variables in WCSP can only take one and only one value from its domain. Integer linear programs are also used in situations where it is only meaningful to make integral quantities in combinatorial optimization problems. However it can be NP-hard to solve a integer linear program in general. By *linear relaxation* [49], the integrality requirement is removed and the integer linear program is solved as a linear program where linear programs were shown to be polynomially solvable. Since linear relaxation enlarges the set of feasible solutions, solving an integer linear program with linear relaxation provides a lower bound on its minimum.

IBM ILOG CPLEX Optimizer [21] is the solver we use in our experiments to solve the integer linear programs with linear relaxation. While the simplex algorithm used by default is not bounded by polynomial time, its excellent average case complexity still allow us to solve the problem efficiently compared to other polynomial time bounded algorithms like the interior point method.

Chapter 4

Polynomially Linear Projection-Safe Cost Functions

Tractable global cost functions require their minimum costs to be computed efficiently. Examples are flow-based projection-safe cost functions and polynomially decomposable cost functions and they can be used practically in WCSPs. However, there are many useful global cost functions which do not have efficient algorithms to compute their minimum costs yet. In this chapter, we first show that the minimum cost of a global cost function can be computed efficiently only if enforcing GAC on its related global constraint is efficient. Accordingly, we prove that it is NP-hard to compute the minimum costs of several useful global cost functions, including the soft variants of SLIDINGSUM, EGCC, and DISJUNCTIVE/CUMULATIVE. To handle such global cost functions in WCSP, we propose another class of $\mathcal{T}$ projection-safe global cost functions called *polynomially linear projection-safe (PLPS)* cost functions. A PLPS cost function W_S can be modeled as an integer linear program whose size is polynomial to the number of variables and the maximum domain size of W_S. First, we give necessary conditions for cost functions to be PLPS. Second, we show that we can efficiently approximate a strong lower bound of the minimum costs of PLPS. We define relaxed consistencies with the approximated minimum costs. We also show that we can conjoin PLPS cost functions easily given their special structures. Third, we give examples of the cost functions which can be

modeled as PLPS cost functions. We demonstrate the efficiency of our approaches experimentally.

4.1 Non-tractable Global Cost Functions in WCSPs

The use of cost functions in WCSPs was limited to binary and ternary cost functions since they were represented as tables, which the time and space requirements in enforcing different consistencies increase exponentially as the numbers of variables in the scope of the cost functions increase. The practical use of global cost functions with high arities is suggested by Lee and Leung [28, 30], who define *flow-based projection-safe global cost functions*. Lee *et al.* [31] further define *tractable projection-safe global cost functions*, which ensure the enforcement of different consistencies on such cost functions in WCSPs is *tractable* and can be done in polynomial time. Given a cost function W_S, W_S is *tractable* if its minimum cost can be found in polynomial time; W_S is a *tractable projection-safe global cost function* if both W_S and W'_S are tractable, where W'_S is W_S after a series of projections or extensions. Lee *et al.* [31] also define *polynomially-decomposable cost functions*, as well as flow-based projection-safe cost functions, belong to the class of tractable projection-safe global cost functions which can be used in WCSPs efficiently.

Given a tuple $\ell \in \mathcal{L}(S)$ in classical CSP, a hard constraint $C_S(\ell)$ returns either it is satisfied or violated. A hard constraint is *tractable* if a tuple satisfying C_S can be found in polynomial time if there exists a such tuple, else the violation can be proven in polynomial time. Given a *hard constraint* C_S and its soft variant W_S, C_S is tractable if W_S is tractable. The tractability of C_S can be shown by computing the minimum cost of the cost function W_S, which can be done in polynomial time. W_S returns 0 if there exists a tuple satisfying C_S, else we can prove that there is no tuple satisfying C_S.

An important consistency technique used in CSP is *generalized arc consistency* (GAC). Here we give the definition of GAC in CSP.

Definition 4.1. *Given a CSP $P(\mathcal{X}, \mathcal{D}, \mathcal{C})$.*

- *A constraint $C_S \in \mathcal{C}$ is* generalized arc consistent *(GAC) if for every value $v_i \in D(x_i)$ and for every $x_i \in S$, there exists a tuple $\ell \in \mathcal{L}(S)$ such that $\ell[x_i] = v_i$ and ℓ satisfies C_S.*
- *P is GAC iff all constraints $C_S \in \mathcal{C}$ are GAC.*

If a hard constraint C_S is tractable, enforcing GAC on C_S must be tractable, as the determinant step of enforcing GAC on C_S is to find supports, which amounts to finding satisfying tuples of C_S.

Efficient enforcement of GAC on a hard constraint in CSPs is required for enforcing consistencies on its soft variant in WCSPs efficiently. However, there are hard constraints which are NP-hard to enforce GAC on them. Currently, there is no known polynomial time algorithm to enforce GAC on such constraints. Here we call these hard constraints *non-tractable constraints*. Similarly, we call the cost functions which are NP-hard to find their minimum costs *non-tractable cost functions* and we have the following lemma.

Lemma 4.2. *Given a non-tractable constraint C_S which is NP-hard to enforce GAC on C_S, and a cost function W_S which is a soft variant of C_S. It is NP-hard to compute the minimum cost of W_S, so W_S must be a non-tractable cost function.*

Proof. We can reduce the problem of either finding a satisfying tuple of a constraint C_S or enforcing GAC (generalized arc consistency) on C_S to the minimum cost computation of the corresponding cost function W_S. Since W_S is a soft variant of C_S, suppose we are given a tuple ℓ, $W_S(\ell)$ returns a cost of 0 if ℓ satisfies C_S. So by computing the minimum cost of W_S, we can know if C_S consists of satisfying tuples. If there exists a satisfying tuple, it can be obtained by repeating the steps for n times, where n equals to the number of variables in C_S.

The determinant step of enforcing GAC on C_S is to find supports, which amounts to finding satisfying tuples of C_S. □

There are many useful cost functions which are non-tractable since they are derived from non-tractable hard constraints. We give an example with the soft variant of the SLIDINGSUM constraint. The SLIDINGSUM constraint is a conjunction of multiple SUM constraints, where the SUM(S, l, u) constraint restricts the sum of the values taken by a set of variables S between a lower bound l and an upper bound u [7].

The SLIDINGSUM$(S, [p_1, \ldots, p_m])$ [34] constraint takes a sequence of n variables $S = \{x_1, \ldots, x_n\}$ and m windows. For every window $p_i = \{l_i, u_i, S_i\}$, the sum of the variables in the set S_i is restricted between a lower bound l_i and an upper bound u_i.

Definition 4.3. *The* SLIDINGSUM*(*$S, [p_1, \ldots, p_m]$*) constraint holds iff*

$$l_i \leq \sum_{x_j \in S_i} x_j \leq u_i$$

for every i *from* 1 *to* m*.*

We can define the SOFT_SLIDINGSUM^{dec}() cost function with the *decomposition-based* violation measure dec by measuring the violation of each window and adding up their costs, which is similar to the one given by Bessière *et. al.* [34].

Definition 4.4. *Given the* SLIDINGSUM*() constraint and an assignment tuple* ℓ *on variables* S*, the soft variant* SOFT_SLIDINGSUM^{dec}*() is defined as:*

$$\text{SOFT\_SLIDINGSUM}^{dec}(S, [p_1, \ldots, p_m])(\ell)$$

$$= \sum_{i=1}^{m} \max(\sum_{x_j \in S_i} \ell[x_j] - u_i, l_i - \sum_{x_j \in S_i} \ell[x_j], 0)$$

Theorem 4.5. *Computing the minimum cost of* SOFT_SLIDINGSUM^{dec} *is NP-hard.*

Proof. Enforcing GAC on a SUM constraint is NP-hard [7]. As the SLIDINGSUM constraint can be represented by a conjunction of multiple SUM constraints, enforcing GAC on SLIDINGSUM is NP-hard. As SOFT_SLIDINGSUM^{dec} is derived

from the SLIDINGSUM constraint, by Lemma 4.2, computing the minimum cost of SOFT_SLIDINGSUM^{dec} is NP-hard. □

There are also cost functions which are not yet proven to be non-tractable and some of them have exponential time algorithms to find their minimum costs. Surely we want some ways to handle such cost functions in WCSPs while they lack efficient algorithms and we propose *Polynomially Linear Projection-Safe (PLPS)* cost functions which give the following results. First, PLPS cost functions are cost functions that can be modeled as integer linear programs with sizes polynomial to the number of variables and the maximum domain size. This class of cost functions has a strong modeling power but it can be NP-hard to compute their minimum costs due to the complexity of solving integer linear programs. Second, we define relaxed consistencies that approximated minimum costs are used instead of exact minimum costs. The approximated minimum costs of PLPS cost functions can be obtained by solving the integer linear programs with linear relaxation in polynomial time. Third, we show that PLPS cost functions can be conjoined easily by conjoining their corresponding linear programs. Our experimental results demonstrate improvements in terms of runtime and search space in general. We give some examples of global cost functions which can be modeled as linear projection-safe cost functions and use experiments to show that our framework allows those global cost functions to be used in WCSPs more efficiently than the existing ways.

4.2 Polynomially Linear Projection-Safe Cost Functions

Linear cost functions are cost functions that can be represented by integer linear programs while their useful properties are preserved after projections and extensions. We first give the definition of a linear cost function.

Definition 4.6. *A cost function W_S is* linear *if it can be represented by an integer linear program I_{W_S}, such that* $\min\{W_S\}$ *is equal to the minimum of I_{W_S}.*

We take the SOFT_SLIDINGSUM cost function mentioned above as an example. Given a SOFT_SLIDINGSUM cost function W_S, we can construct the corresponding integer linear program I_{W_S} so that the SOFT_SLIDINGSUM cost function is a linear cost function.

Given a cost function W_S, we create a variable c_{x_i} in I_{W_S} for each variable $x_i \in S$ which has the same domain as x_i such that $c_{x_i} = x_i$. Two set of variables $L = \{L_1, \ldots, L_m\}$ and $U = \{U_1, \ldots, U_m\}$ are introduced to represent the cost arising from violating the related hard constraint if the sum of the values is smaller than the lower bound or greater than the upper bound respectively.

Theorem 4.7. *The* SOFT_SLIDINGSUM^{dec} *cost function is a linear cost function.*

Proof. The SOFT_SLIDINGSUM$^{dec}(S, [p_1, \ldots, p_m])$ cost function can be expressed as an integer linear program I where I is defined as:

$$\begin{array}{cc} \min \sum_{j=1}^{m} L_j + U_j & \text{s.t.} \\ l_j \leq \sum_{x_i \in S_j} c_{x_i} - L_j + U_j \leq u_j & \forall j = 1 \ldots m \\ L_j \geq 0, U_j \geq 0 & \forall j = 1 \ldots m \\ c_{x_i} = \{D_{x_i}\} & \forall x_i \in S \end{array}$$

The minimum of I gives the minimum cost of SOFT_SLIDINGSUM$^{dec}(S, [p_1, \ldots, p_m])$. By Definition 4.6, the SOFT_SLIDINGSUM^{dec} cost function is a linear cost function. □

Koster [23] suggests a method to formulate global cost functions into integer linear programs by treating them as table cost functions and modeling the cost of each tuple by an inequality.

Theorem 4.8. *[23] Given a cost function W_S, where $\mathcal{L}(S)$ is a set of tuples corresponding to all possible assignments on the set of variables S, $\ell \in \mathcal{L}(S)$ is tuple*

represents an assignment, $\ell[x]$ denotes the value assigned to x in ℓ and $W_S(\ell)$ returns the cost of the tuple ℓ in W_S. W_S is linear since the corresponding integer linear program can be defined as:

$$\min \sum_{\ell \in \mathcal{L}(S)} W_S(\ell) * b_\ell$$

$$\sum_{\ell[x]=a, \ell \in \mathcal{L}(S)} b_\ell - c_{x,a} = 0, \quad \forall a \in D(x), x \in S$$

$$\sum_{a \in D(x)} c_{x,a} = 1, \quad \forall x \in D(x)$$

$$c_{x,a} \in \{0,1\}, \quad \forall a \in D(x), x \in S$$

$$b_\ell \in \{0,1\}, \quad \forall \ell \in \mathcal{L}(S)$$

By this method, we can model every cost function into a linear cost function. However, the number of linear inequalities used is exponential to the number of variables in the cost function, which is undesirable if we are looking for efficient ways to solve them. In this thesis, we are focusing on a special class of linear cost functions which the size of their corresponding integer linear programs are polynomial to size of the cost functions, and we define *polynomially linear* cost functions.

Definition 4.9. *Suppose W_S is a cost function. W_S is* polynomially linear *if I_{W_S} has the number of inequalities and the number of variables polynomial to the number of variables and the maximum domain size of W_S, where I_{W_S} is the corresponding integer linear program of W_S.*

A $\mathcal{T}$ projection-safe cost function preserves its property $\mathcal{T}$ after projections and extensions. For example, $\mathcal{T}$ can be flow-based and Lee and Leung [28, 30] give examples of flow-based projection-safe cost functions. If the minimum cost of a $\mathcal{T}$ projection-safe cost function can be computed efficiently, its minimum cost can still

be computed efficiently through the consistency enforcements, so it is feasible to use such a cost function in WCSPs.

We are interested in using *polynomially linear* as the property $\mathcal{T}$ and we define *polynomially linear projection-safe* (PLPS) cost functions.

Definition 4.10. *Suppose W_S is a polynomially linear cost function. W_S is* polynomially linear projection-safe *if W'_S is also polynomially linear projection-safe, where W'_S is W_S after a series of projections and extensions.*

First we give the sufficient conditions to determine whether a cost function is a PLPS cost function. Then we show that given a PLPS cost function, the minimum cost can still be computed by solving its corresponding integer linear program after projections and extension.

Lemma 4.11. *Given a cost function W_S which satisfies the following three conditions:*

1. *W_S is linear and has the corresponding integer linear program I_{W_S}, where the number of inequalities and number of variables of I_{W_S} are polynomial to the number of variables and the maximum domain size of W_S;*

2. *there exists a surjective function Λ' mapping each optimal feasible solution $\gamma_{I_{W_S}}$ in I_{W_S} to each tuple $\ell[S] \in \mathcal{L}(S)$, where $\mathcal{L}(S)$ denotes the set of tuples corresponding to all possible assignments on variables S, and;*

3. *for each value $v \in D(x_i)$ in each variable $x_i \in S$, there exists an injection mapping an assignment $\{x_i \mapsto v\}$ to a 0-1 variable $c_{x_i,v}$ in I_{W_S} such that if $\ell[S] = \Lambda'(\gamma_{I_{W_S}})$ for an optimal solution $\gamma_{I_{W_S}}$ in I_{W_S} and a tuple $\ell[S] \in \mathcal{L}$, whenever $\ell[x_i] = v$ for some tuple $\ell[S]$, $\gamma_{I_{W_S}}[c_{x_i,v}] = 1$; whenever $\ell[x_i] \neq v$, $\gamma_{I_{W_S}}[c_{x_i,v}] = 0$*

Suppose W'_S is obtained from projecting α from W_S to $W_{x_i}(v)$, or extending α from $W_{x_i}(v)$ to W_S, then W'_S also satisfies these conditions.

Proof. Assume W_S is a PLPS cost function and I_{W_S} is the corresponding integer linear program of W_S. We first consider the part of projection, *i.e.* W'_S is defined as:

$$W'_S(\ell) = \begin{cases} W_S(\ell) \ominus \alpha & \text{if } \ell[x_i] = v \\ W_S(\ell) & \text{otherwise} \end{cases}$$

We first show that W'_S is also a linear cost function with polynomial size (condition 1)). After projection, we can construct a new integer linear program $I_{W'_S}$ from I_{W_S} by adding an additional term $-\alpha c_{i,v}$ to the objective function of I_{W_S}. The resulting integer linear program $I_{W'_S}$ is corresponding to W'_S, since:

$$\begin{aligned} \min(I_{W'_S}) &= \min(I_{W_S}) \ominus \alpha c_{i,v} \\ &= \min\{W_S\} \ominus \alpha c_{i,v} \\ &= \begin{cases} \min\{W_S\} \ominus \alpha & \text{, if } c_{i,v} = 1 \\ \min\{W_S\} & \text{, if } c_{i,v} = 0 \end{cases} \\ &= \min\{W'_S\}. \end{aligned}$$

Thus, W'_S is linear with the corresponding integer linear program $I_{W'_S}$ and satisfies the condition 1). Moreover, since $I_{W'_S}$ has the same set of variables and linear inequalities as I_{W_S} has, W'_S also satisfies the conditions 2) and 3).

Then we consider the part of extension, *i.e.* W'_S is defined as:

$$W'_S(\ell) = \begin{cases} W_S(\ell) \oplus \alpha & \text{if } \ell[x_i] = v \\ W_S(\ell) & \text{otherwise} \end{cases}$$

After extension, we can construct a new integer linear program $I_{W'_S}$ from I_{W_S} by adding an additional term $+\alpha c_{i,v}$ to the objective function of I_{W_S}.

With similar arguments, the new integer linear program $I_{W'_S}$ is still corresponding to W'_S. Thus W'_S satisfies the condition 1). Moreover, since $I_{W'_S}$ has the same set of variables and linear inequalities as I_{W_S} has, W'_S also satisfies the conditions

2) and 3). □

Lemma 4.11 implies that if a linear cost function satisfies conditions 2) and 3), those conditions are preserved throughout a series of projections and extensions. From Lemma 4.11, we can give the sufficient conditions of a PLPS cost function.

Theorem 4.12. *If a global cost function W_S satisfies the conditions stated in Lemma 4.11, it is a PLPS cost function.*

Proof. From Lemma 4.11, W_S preserves the conditions, as well as the linearity, throughout a series of projection and extension operations. By the definitions of $\mathcal{T}$ projection-safe cost functions, W_S is a PLPS cost function. □

Theorem 4.12 gives a sufficient condition for a global cost function to be a PLPS cost function. In order to construct the corresponding integer linear program I_{W_S} such that the conditions of a PLPS cost function can be satisfied, binary variables $c_{x_i,d}$ are introduced for every value d in the domain $d \in D(x_i)$ of every variable $x_i \in S$ in I_{W_S}; for each variable $x_i \in S$, there is an extra linear cost function $\sum_{j \in D(x_i)} c_{x_i,j}$ added to I_{W_S} such that only a value can be assigned to each variable x_i in W_S. According to condition 3), we can easily define Λ'. In addition, the proof part of Lemma 4.11 demonstrates a general procedure of performing projections and extensions on PLPS cost functions.

We use the SOFT_SLIDINGSUM cost function as an example of a PLPS cost function. Then, we use this cost function to give another example, which demonstrates how costs can be projected (and extended) to PLPS cost functions while the linear projection-safety is preserved.

Theorem 4.13. *The* SOFT_SLIDINGSUM^{dec} *cost function is a PLPS cost function.*

Proof. The SOFT_SLIDINGSUM$^{dec}(S, [p_1, \ldots, p_m])$ cost function can be expressed

as an integer linear program I_{W_S} where I_{W_S} is defined as:

$$\begin{array}{cc} \min \sum_{j=1}^{m} L_j + U_j & \text{s.t.} \\ l_j \leq \sum_{x_h \in S_j} \sum_{d \in D(h)} d * c_{x_h,d} - L_j + U_j \leq u_j & \forall j = 1 \ldots m \\ L_j \geq 0, U_j \geq 0 & \forall j = 1 \ldots m \\ \sum_{d \in D(x_i)} c_{x_i,d} = 1 & \forall i = 1 \ldots n \\ c_{x_i,d} \in \{0,1\} & \forall x_i \in S, d \in D(x_i) \end{array}$$

Let $D_{\max}$ be the maximum domain size for the variables in S, the corresponding integer linear program uses $|S| * D_{\max} + 2 * m$ variables and $3 * m + |S| + |S| * D_{\max}$ inequalities. If $x_i = d$, $c_{x_i,d} = 1$; otherwise $c_{x_i,d} = 0$. By Theorem 4.12, SOFT_SLIDINGSUM^{dec} cost function is a PLPS cost function. □

Example 4.14. *Consider the following WCSP $P(\mathcal{X}, \mathcal{D}, W_S, k)$:*

$\mathcal{X} = \{x_1, x_2, x_3\}$, $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3\}$, $p_1 = \{3, 4, \{x_1, x_2\}\}$, $p_2 = \{4, 5, \{x_2, x_3\}\}$, $W_S =$ SOFT_SLIDINGSUM$^{dec}([x_1, x_2, x_3], [p_1, p_2])$. *The corresponding integer linear program of W_S is:*

$$\min L_1 + U_1 + L_2 + U_2 \; s.t.$$

$$3 \leq c_{x_1,1} + 2c_{x_1,2} + 3c_{x_1,3} + c_{x_2,1} + 2c_{x_2,2} + 3c_{x_2,3} - L_1 + U_1 \leq 4$$

$$4 \leq c_{x_2,1} + 2c_{x_2,2} + 3c_{x_2,3} + c_{x_3,1} + 2c_{x_3,2} + 3c_{x_3,3} - L_2 + U_2 \leq 5$$

$$c_{x_1,1} + c_{x_1,2} + c_{x_1,3} = 1$$

$$c_{x_2,1} + c_{x_2,2} + c_{x_2,3} = 1$$

$$c_{x_3,1} + c_{x_3,2} + c_{x_3,3} = 1$$

$$L_1 \geq 0, U_1 \geq 0, L_2 \geq 0, U_2 \geq 0$$

where $c_{x_1,1}, c_{x_1,2}, c_{x_1,3}, c_{x_2,1}, c_{x_2,2}, c_{x_2,3}, c_{x_3,1}, c_{x_3,2}, c_{x_3,3} \in \{0, 1\}$.

Suppose a cost of 2 is projected from W_S to $W_{x_1}(1)$ such that a term is added to

the objective function of the corresponding integer linear program of W_S*. Since the other parts of the corresponding integer linear program of* W_S *remain unchanged,* W_S *is still PLPS. The corresponding integer linear program of* W_S *becomes:*

$$\min L_1 + U_1 + L_2 + U_2 - 2c_{x_1,1} \; s.t.$$

$$3 \leq c_{x_1,1} + 2c_{x_1,2} + 3c_{x_1,3} + c_{x_2,1} + 2c_{x_2,2} + 3c_{x_2,3} - L_1 + U_1 \leq 4$$

$$4 \leq c_{x_2,1} + 2c_{x_2,2} + 3c_{x_2,3} + c_{x_3,1} + 2c_{x_3,2} + 3c_{x_3,3} - L_2 + U_2 \leq 5$$

$$c_{x_1,1} + c_{x_1,2} + c_{x_1,3} = 1$$

$$c_{x_2,1} + c_{x_2,2} + c_{x_2,3} = 1$$

$$c_{x_3,1} + c_{x_3,2} + c_{x_3,3} = 1$$

$$L_1 \geq 0, U_1 \geq 0, L_2 \geq 0, U_2 \geq 0$$

where $c_{x_1,1}, c_{x_1,2}, c_{x_1,3}, c_{x_2,1}, c_{x_2,2}, c_{x_2,3}, c_{x_3,1}, c_{x_3,2}, c_{x_3,3} \in \{0, 1\}$.

Linear relaxation allows the minimum of the corresponding integer linear programs of PLPS cost functions to be approximated in polynomial time. Accordingly, the relaxed consistency notions can be defined, which are weaker but can be enforced more efficiently.

4.3 Relaxed Consistencies on Polynomially Linear Projection-Safe Cost Functions

Polynomially linear projection-safe (PLPS) cost functions can be represented as *integer linear programs* [49]. It is NP-hard to solve an integer linear program in general, but a good approximation of the minimum cost can be computed with the linear relaxation using linear programming. Given a PLPS cost function W_S and its corresponding integer linear program I_{W_S}, we first define the the value of the

objective function z from an optimal feasible solution of $z = \min c^T X$ using linear relaxation as $\text{relaxed\_min}(I_{W_S})$. We have the following theorem according to the properties of linear relaxation.

Theorem 4.15. *[49] Given an integer linear program* I_{W_S}, $\text{relaxed\_min}(I_{W_S}) \leq \min(I_{W_S})$ *and* $\lceil \text{relaxed\_min}(I_{W_S}) \rceil \leq \min(I_{W_S})$.

The pair of $\lceil\ \rceil$ symbols represents the ceiling function, where $\lceil x \rceil$ gives the smallest integer not less than x.

Given a PLPS cost function W_S and its corresponding integer linear program I_{W_S}, solving I_{W_S} by linear relaxation gives an lower bound of its minimum cost $\min\{W_S\}$. Such an approximation of the minimum costs by linear relaxation forms the basis of relaxed but weaker forms of common consistencies for PLPS cost functions. We name the approximation of the minimum costs of a PLPS cost function W_S by solving its corresponding integer linear program I_{W_S} with linear relaxation $\text{relaxed\_min}(I_{W_S})$ as *relaxed minimum costs* denoted as $\text{relaxed\_min}\{W_S\}$, such that $\text{relaxed\_min}\{W_S\} = \text{relaxed\_min}(I_{W_S})$. Since $\min(I_{W_S}) = \min\{W_S\}$, we have the following corollary:

Corollary 4.16. *Given a PLPS* W_S *and its corresponding integer linear program* I_{W_S}, $\text{relaxed\_min}\{W_S\} \leq \min\{W_S\}$ *and* $\lceil \text{relaxed\_min}\{W_S\} \rceil \leq \min\{W_S\}$.

To define a relaxed version of GAC* using the relaxed minimum costs of PLPS cost functions, we first reformulate the definition of GAC*. GAC* requires that for each value of each variable, there must exists a supporting tuple with its cost equals to 0 in each cost function related to that value of that variable. If $\min\{W_S(\ell) | \ell[x_i] = a)\} = 0$, there exists such a supporting tuple for the value a in the variable x_i. So we give an equivalent definition of GAC* according to Definition 3.8:

Definition 4.17. *A variable* $x_i \in S$ *is* GAC* *[15] with respect to a cost function* W_S *if:*

- x_i *is NC*, and;*

- *for each value* $v_i \in D(x_i)$, $\min\{W_S(\ell) \mid \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} = 0$.

A WCSP is GAC* *iff all variables are GAC* with respect to all cost functions in* $\mathcal{C}$.

By Corollary 4.16, we can define an relaxed version of GAC* called *relaxed GAC** by relaxing the requirements of GAC* and replacing $\min\{W_S\}$ by $\text{relaxed\_}\min\{W_S\}$.

Definition 4.18. *A variable* $x_i \in S$ *is* relaxed GAC* *with respect to a cost function* W_S *if:*

- x_i *is NC*, and;*
- *for each value* $v_i \in D(x_i), \text{relaxed\_}\min\{W_S(\ell) \mid \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} \leq 0$. *of* v_i *with respect to* C_S.

To compare the strength of GAC* and relaxed GAC*, we define that given a WCSP P, a consistency α is *strictly weaker* than another consistency β, written as $\alpha < \beta$, iff P is α whenever P is β, but not vice versa. Since $\text{relaxed\_}\min\{W_S\}$ is a lower bound of $\min\{W_S\}$, by Corollary 4.16 we immediately have the following theorem.

Theorem 4.19. *Relaxed GAC* is strictly weaker than GAC*.*

According to the algorithm of enforcing GAC*, any WCSP can be transformed to an equivalent one which is GAC*. Here we give the algorithm of enforcing relaxed GAC* which can transform any WCSP to an equivalent one which is relaxed GAC*. It is similar to that of enforcing GAC* listed in Algorithm 7, except that $\text{relaxed\_}\min(W_S)$ does not always return an integer. We define the cost to be projected in enforcing relaxed GAC* $\alpha' = \max(\lceil \text{relaxed\_}\min\{W_S\} \rceil, 0)$ and we have the following theorem.

Theorem 4.20. *Suppose* I_{W_S} *is an integer linear program corresponding to a PLPS cost function* W_S*, and there exists a cost* $\alpha = \min\{W_S\}$ *to be projected in enforcing GAC*. After projecting a cost* $\alpha' = \max(\lceil \text{relaxed\_}\min\{W_S\} \rceil, 0)$ *in enforcing relaxed GAC*,* $\min\{W_S\}$ *is greater than or equal to 0.*

Proof. Let $\min(I_{W_S}) = \alpha$, such that after projecting α in enforcing GAC*, $\min\{W_S\}$ is greater than or equal to 0. Solving I_{W_S} by linear relaxation obtains an relaxed minimum cost $\text{relaxed\_min}\{W_S\} = \text{relaxed\_min}(I_{W_S})$ to be projected. Given that $\alpha' = \lceil \text{relaxed\_min}(I_{W_S}) \rceil \leq \min(I_{W_S}) = \alpha$, we can ensure that after projecting $\lceil \text{relaxed\_min}(I_{W_S}) \rceil$, $\min\{W_S\}$ is still greater than or equal to 0.

At the same time as $\min\{W_S\}$ is greater than or equal to 0 after projecting α in enforcing GAC*. Even $\text{relaxed\_min}(I_{W_S}) < 0$ after enforcing relaxed GAC*, we can still ensure that $\min\{W_S\}$ is greater than or equal to 0. □

The procedure `enforceRelaxedGAC*` in Algorithm 7 enforce relaxed GAC* for a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$ based on Algorithm 3 of enforcing GAC*. The function `findSupport()` is replaced by `relaxedFindSupport()` and the cost to be projected α becomes $\max(\lceil \text{relaxed\_min}\{W_S\} \rceil, 0)$.

To define the relaxed version of FDGAC*, we first give an equivalent definition of FDGAC* according to Definition 3.11:

Definition 4.21. *A variable $x_i \in S$ is* DGAC* *[15] with respect to a cost function W_S if:*

- *x_i is NC*, and;*
- *for each value $v_i \in D(x_i)$, $\min\{W_S(\ell) \oplus \bigoplus_{x_j \mid j>i} W_j(\ell[x_j]) \mid \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} = 0$.*

A WCSP is FDGAC iff it is GAC* and all variables are DGAC* with respect to all cost functions in $\mathcal{C}$.*

By Corollary 4.16, we can define an the relaxed version of FDGAC* called *relaxed FDGAC** by relaxing the requirements of FDGAC* and replacing $\min\{W_S\}$ by $\text{relaxed\_min}\{W_S\}$.

Definition 4.22. *A variable $x_i \in S$ is* relaxed DGAC* *with respect to a cost function W_S if:*

```
1  Procedure enforceRelaxedGAC*()begin
2      Q := X;
3      while Q ≠ ∅ do
4          x_j :=pop(Q);
5          flag := false;
6          foreach W_S s.t. {x_j} ⊂ S do
7              foreach x_i ∈ S \ {x_j} do
8                  flag := flag ∨ relaxedFindSupport(W_S, x_i);
9                  if pruneVal(x_i) then Q := Q ∪ {x_i};
10         if flag then
11             foreach x_i ∈ X do
12                 if pruneVal(x_i) then Q := Q ∪ {x_i};
13 Function relaxedFindSupport(W_S, x_i):Boolean begin
14     flag := false;
15     foreach v ∈ D(x_i) do
16         α := max(⌈relaxed_min{W_S(ℓ)|ℓ ∈ L(S) ∧ ℓ[x_i] = v}⌉, 0);
17         if W_i(v) = 0 ∧ α > 0 then flag := true;
18         W_i(v) := W_i(v) ⊕ α;
19         foreach ℓ ∈ L(S) s.t. ℓ[x_i] = v do
20             W_S(ℓ) := W_S(ℓ) ⊖ α;
21     unaryProject(x_i);
22     return flag;
```

Algorithm 7: Enforcing relaxed GAC* for a WCSP

- x_i *is NC*, and;*
- *for each value* $v_i \in D(x_i)$, $\text{relaxed\_min}\{W_S(\ell) \oplus \bigoplus_{x_j|j>i} W_j(\ell[x_j]) \mid \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} \leq 0$.

A WCSP is relaxed FDGAC iff it is relaxed GAC* and all variables are relaxed DGAC* with respect to all cost functions in* $\mathcal{C}$.

Since $\text{relaxed\_min}\{W_S\}$ is a lower bound of $\min\{W_S\}$, by Corollary 4.16 we immediately have the following theorem.

Theorem 4.23. *Relaxed FDGAC* is strictly weaker than FDGAC*.*

The procedure of enforcing relaxed FDGAC* is similar to that of enforcing FDGAC* in Algorithm 4, except that the `findSupport()` function is replaced by the `relaxedFindSupport()` function in Algorithm 7.

To define a relaxed version of weak EDGAC*, first we give an equivalent definition of weak EDGAC* according to Definition 3.15:

Definition 4.24. *Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$ and any fully supported set $U(W_S, x_i)$ for each variable $x_i \in \mathcal{X}$ and each cost function $W_S \in \mathcal{C}$. A variable $x_i \in S$ is weak EGAC* [29, 30] if:*

- *x_i is NC*, and;*
- *there exists a value $v \in D(x_i)$ such that for each cost function $W_S \in \mathcal{C}$ with $x_i \in S$ and $U(W_S, x_i)$, $\min\{\bigoplus_{x_i \in S} W_S(\ell) \oplus \bigoplus_{x_j | j \in U(W_S, x_i)} W_j(\ell[x_j]) \mid \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} = 0$.*

A WCSP is weak EDGAC iff it is FDGAC* and all variables are weak EDGAC*.*

By Corollary 4.16, we can define an relaxed of relaxed weak EDGAC* called *relaxed weak EDGAC** by relaxing the requirements of weak EDGAC* and replacing $\min\{W_S\}$ by $\text{relaxed\_min}\{W_S\}$.

Definition 4.25. *Given a WCSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$ and any fully supported set $U(W_S, x_i)$ for each variable $x_i \in \mathcal{X}$ and each cost function $W_S \in \mathcal{C}$. A variable $x_i \in S$ is relaxed weak EGAC* if:*

- *x_i is NC*, and;*
- *there exists a value $v \in D(x_i)$ such that for each cost function $W_S \in \mathcal{C}$ with $x_i \in S$ and $U(W_S, x_i)$, $\text{relaxed\_min}\{\bigoplus_{x_i \in S} W_S(\ell) \oplus \bigoplus_{x_j | j \in U(W_S, x_i)} W_j(\ell[x_j]) \mid \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v_i\} \leq 0$.*

A WCSP is relaxed weak EDGAC iff it is relaxed FDGAC* and all variables are relaxed weak EDGAC*.*

Since $\text{relaxed\_min}\{W_S\}$ is a lower bound of $\min\{W_S\}$, by Corollary 4.16 we immediately have the following theorem.

Theorem 4.26. *Relaxed weak EDGAC* is strictly weaker than weak EDGAC*.*

The procedure of enforcing relaxed weak EDGAC* is similar to that of enforcing weak EDGAC* in Algorithm 6, except that the `findSupport()` function in the `findFullSupport()` function is replaced by the `relaxedFindSupport()` function in Algorithm 7, similar to the algorithm of enforcing relaxed FDGAC*.

4.4 Conjoining Polynomially Linear Projection-Safe Cost Functions

If two constraints or cost functions share more than one variable, they are *overlapping*. In the rest parts of this thesis, we consider conjunctions of overlapping cost functions. In general, enforcing a consistency on the individual cost functions may not imply the same consistency on the conjunction of the two. An example is given by Bessière *et al.* [6]. According to that example, enforcing GAC on two overlapping ALLDIFF constraints does not imply GAC on the conjunction of them. It is easy to check that the result also holds for cost functions. By discovering extra pruning opportunities, propagating on conjunctions of cost functions may reduce more search space than propagating on individual cost functions can.

Every PLPS cost function has an associated integer linear program. PLPS cost functions can be conjoined together easily by combining their corresponding integer linear programs in a straightforward manner. Given two integer linear programs $I_{W_{S_1}}$ and $I_{W_{S_2}}$, we define $I_{W_{S_1}} \wedge I_{W_{S_2}}$ to be their combination by taking the union of their linear inequalities and adding up their objective functions. The following theorem ensures that conjunctions of PLPS cost functions remain PLPS.

Lemma 4.27. *Suppose W_{S_1} and W_{S_2} are PLPS cost functions. The conjunction $W_{conj} \equiv W_{S_1} \wedge W_{S_2}$ is also PLPS.*

Proof. Suppose W_{S_1} and W_{S_2} have their corresponding integer linear program $I_{W_{S_1}}$ and $I_{W_{S_2}}$ respectively. The integer linear program $I_{W_{conj}}$ for W_{conj} can simply be formed by $I_{W_{conj}} \equiv I_{W_{S_1}} \wedge I_{W_{S_2}}$. It is easy to check that W_{conj} satisfies the conditions for PLPS. □

An immediate question is whether a conjunction of PLPS cost functions always gives a stronger bound than using the individual PLPS cost functions, given that the same level of consistency is maintained. Given WCSP $P_{PLPS} = (\mathcal{X}, \mathcal{D}, \mathcal{C}_{PLPS}, k)$, where each cost function $W_S \in \mathcal{C}_{PLPS}$ is PLPS with corresponding integer linear program I_{W_S}. We assume that $\mathcal{C}_{PLPS}$ contains overlapping cost functions. We can construct an equivalent WCSP $P_{conj} = (\mathcal{X}, \mathcal{D}, \mathcal{C}_{conj}, k)$ where $\mathcal{C}_{conj} = \{W_{conj}\}$ and $W_{conj} \equiv \bigwedge_{W_S \in \mathcal{C}_{PLPS}} W_S$ with the corresponding scope $S_{conj} \equiv \bigcup_{W_S \in \mathcal{C}_{PLPS}} S$ and integer linear program $I_{W_{conj}} \equiv \bigwedge_{W_S \in \mathcal{C}_{PLPS}} I_{W_S}$. Since $\mathcal{C}_{PLPS}$ is a set of PLPS cost functions, the conjunction W_{conj} must be a PLPS cost function.

Given a problem P representable by two WCSP models $\phi(P)$ and $\psi(P)$. A consistency Φ on $\phi(P)$ is *strictly stronger* than another consistency Ψ on $\psi(P)$, written as Φ on $\phi(P) > \Psi$ on $\psi(P)$, iff $\psi(P)$ is Ψ whenever $\phi(P)$ is Φ, but not *vice versa* [30].

We show that (FD)GAC* and weak EDGAC* on P_{conj} are strictly stronger than their counterparts on P_{PLPS} respectively by the following theorem.

Theorem 4.28. *Suppose α-consistency is one of GAC*, FDGAC*, and weak EDGAC*. We have α-consistent on P_{conj} > α-consistent on P_{PLPS}.*

Proof. We prove the part for GAC*. The proofs for the other consistencies are similar.

Assume P_{conj} is GAC*, but P_{PLPS} is not GAC*. There exists a variable $x_i \in \mathcal{X}$ with a value $a \in D(x_i)$ and a cost function $W_S \in \mathcal{C}_{PLPS}$ in P_{PLPS} such that

$\min\{W_S(\ell) \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S)\} > 0$. Now, we have

$$\begin{aligned} & \min\{W_{conj} \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S_{conj})\} \\ \geq \; & \bigoplus_{W_S \in \mathcal{C}_{PLPS}} \min\{W_S(\ell) \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S)\} > 0 \end{aligned}$$

So we cannot find a simple support for a and x_i cannot be GAC* with respect to W_{conj} in P_{conj}.

Consider $W_{S_1} = \text{SOFT\_ALLDIFF}^{var}(x_1, x_2, x_3)$ and $W_{S_2} = \text{SOFT\_ALLDIFF}^{var}(x_2, x_3, x_4)$, where $D(x_1) = \{a, b\}$, $D(x_2) = D(x_3) = \{a, b, c\}$ and $D(x_4) = \{b, c\}$. It is easy to check that $P_{PLPS} = (\mathcal{X}, \mathcal{D}, \{W_{S_1}, W_{S_2}\}, k)$ is GAC*. However, $P_{conj} = (\mathcal{X}, \mathcal{D}, \{W_{S_1} \wedge W_{S_2}\}, k)$ is not GAC* since the minimum cost when $x_1 = a$ is $1 > 0$.

Result follows. □

When standard consistencies are replaced by their relaxed versions, result similar to that of Theorem 4.28 does not hold. For simplicity, we assume $\mathcal{C}_{PLPS} = \{W_{S_1}, W_{S_2}\}$. Suppose P_{conj} is relaxed GAC*. We have

$$\begin{aligned} 0 \; \geq \; & \text{approx\_min}\{W_{conj} \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S_{conj})\} \\ \geq \; & \bigoplus_{W_S \in \mathcal{C}_{PLPS}} \text{approx\_min}\{W_S(\ell) \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S)\} \end{aligned}$$

In order for the sum to be non-positive, it is possible for the approximated minimum cost of one of $\{W_{S_1}, W_{S_2}\}$ to be negative and the other one positive. Therefore, one of them is not relaxed GAC*. However, this peculiar bad situation just described does not happen often in practice and we will demonstrate that it is worthwhile to propagate on the conjunction instead of individual cost functions in the experiments in the last section of this chapter.

4.5 Modeling Global Cost Functions as Polynomially Linear Projection-Safe Cost Functions

In this section we introduce three global cost functions, including the SOFT_SLIDINGSUM, SOFT_EGCC, and SOFT_DISJUNCTIVE/CUMULATIVE cost functions. Following the Lemma 4.2, we prove that it is NP-hard to compute their minimum cost by showing that it is NP-hard to enforce GAC [9], a consistency notion in classical CSPs, on the related hard constraint. By modeling them as polynomially linear projection-safe cost functions, we can obtain the relaxed minimum costs by linear relaxation and enforce relaxed consistencies.

4.5.1 The SOFT_SLIDINGSUM^{dec} Cost Function

The SLIDINGSUM() constraint [34] represents a sequence of SUM() constraints and each of the SUM() constraint restricts the sum of the values taken by the variables in its scope between between a lower bound and an upper bound. A soft variant for the SLIDINGSUM() constraint is called the SOFT_SLIDINGSUM^{dec}() cost function. The definition of SOFT_SLIDINGSUM^{dec}() is given in Definition 4.4 in Section 4.1 and it is shown to be PLPS in Theorem 4.13 in Section 4.2. Here we give an example of modeling a SOFT_SLIDINGSUM^{dec}() cost function as a PLPS cost function.

Example 4.29. *Consider the following WCSP* $P(\mathcal{X}, \mathcal{D}, W_S, k)$*:*

$\mathcal{X} = \{x_1, x_2, x_3\}$, $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3\}$, $p_1 = \{3, 4, \{x_1, x_2\}\}$, $p_2 = \{4, 5, \{x_2, x_3\}\}$, $W_S =$ SOFT_SLIDINGSUM$^{dec}([x_1, x_2, x_3], [p_1, p_2])$. *For example,* $W_S(1, 1, 3) = 1$ *because* $l_1 - (x_1 + x_2) = 1$ *and* $l_2 \leq (x_2 + x_3) \leq u_2$*. The corresponding integer linear program of* W_S *is:*

$$\min L_1 + U_1 + L_2 + U_2 \; s.t.$$

$$3 \leq c_{x_1,1} + 2c_{x_1,2} + 3c_{x_1,3} + c_{x_2,1} + 2c_{x_2,2} + 3c_{x_2,3} - L_1 + U_1 \leq 4$$

$$4 \leq c_{x_2,1} + 2c_{x_2,2} + 3c_{x_2,3} + c_{x_3,1} + 2c_{x_3,2} + 3c_{x_3,3} - L_2 + U_2 \leq 5$$

$$c_{x_1,1} + c_{x_1,2} + c_{x_1,3} = 1$$

$$c_{x_2,1} + c_{x_2,2} + c_{x_2,3} = 1$$

$$c_{x_3,1} + c_{x_3,2} + c_{x_3,3} = 1$$

$$L_1 \geq 0, U_1 \geq 0, L_2 \geq 0, U_2 \geq 0$$

where $c_{x_1,1}, c_{x_1,2}, c_{x_1,3}, c_{x_2,1}, c_{x_2,2}, c_{x_2,3}, c_{x_3,1}, c_{x_3,2}, c_{x_3,3} \in \{0, 1\}$.

4.5.2 The SOFT_EGCC^{var} Cost Function

The EGCC(S_X, S_Y) constraint [22] is defined for two sets of $n + m$ variables S_X and S_Y where $S_X = \{x_1, \ldots, x_n\}$ is a set of assignment variables and $S_Y = \{y_{d_1}, \ldots, y_{d_m}\}$ is a set of counting variables. The idea is that each value d_j where $y_{d_j} \in S_Y$ is used exactly y_{d_j} times by the variables S_X.

Definition 4.30. *The* EGCC*(S_X, S_Y) constraint holds iff* $occ(d_i, (x_1, \ldots, x_n)) = y_{d_i}$ *for every* d_i *where* $y_{d_i} \in S_Y$.

where $occ(v, \mathcal{T})$ *is the number of occurrences of* v *in* $\mathcal{T}$.

Theorem 4.31. *Enforcing GAC on every variable of* EGCC *is NP-hard [22].*

We can define the SOFT_EGCC^{var} cost function with the same violation measure as the variable-based violation measure used in SOFT_EGCC^{var} [48]. The constraint is softened by allowing the counting variables $y_{d_i} \in S_Y$ to take values other than $occ(d_i(x_1, \ldots, x_n))$.

Definition 4.32. *Given the* EGCC*() constraint and an assignment tuple* ℓ *in variables* $S = S_X \cap S_Y$*,*

$$\text{SOFT\_EGCC}^{var}(S_X, S_Y)(\ell)$$

$$= \sum_{j=1}^{m} |y_{d_j} - occ(d_j, (x_1, \ldots, x_n))|$$

Theorem 4.33. *Computing the minimum cost of* SOFT_EGCC^{var}*() is NP-hard.*

Proof. The SOFT_EGCC^{var}() cost function is derived from the EGCC constraint and it is NP-hard to enforce GAC on the EGCC constraint. So, computing the minimum cost of SOFT_EGCC^{var}() is NP-hard. □

We can model this cost function in the form of a PLPS cost function such that we can compute the approximated minimum cost efficiently by linear relaxation.

Theorem 4.34. *The* SOFT_EGCC^{var}*() cost function is a PLPS cost function.*

Proof. The SOFT_EGCC^{var}() cost function can be expressed as an integer linear program I where I is defined as:

$$
\begin{array}{cc}
\min \sum_{j=1}^{m} L_j + U_j & \text{s.t.} \\
\sum_{i=1}^{n} c_{x_i,d_j} - (\sum_{h \in D(y_{d_j})} h * c_{y_{d_j},h}) - L_j + U_j = 0 & \forall j = 1 \ldots m \\
L_j \geq 0, U_j \geq 0 & \forall j = 1 \ldots m \\
\sum_{j=1}^{m} c_{x_i,d_j} = 1 \quad \forall d_j \in D(x_i) & \forall i = 1 \ldots n \\
\sum_{j=1}^{m} c_{x_i,d_j} = 0 \quad \forall d_j \notin D(x_i) & \forall i = 1 \ldots n \\
\sum_{h \in D_{y_{d_j}}} c_{y_{d_j},h} = 1 & \forall j = 1 \ldots m \\
c_{x_i,d_j} \in \{0,1\} & \forall x_i \in X, d_j \in D(x_i) \\
c_{y_{d_j},h} \in \{0,1\} & \forall y_{d_j} \in Y, h \in D_{y_{d_j}}
\end{array}
$$

Let $D_{\max}$ be the maximum domain size for the variables in S, the corresponding integer linear program uses $|X| * D_{\max} + |Y| * D_{\max} + 2 * D_{\max}$ variables and $4 * |Y| + 2 * |X| + |X| * D_{\max} + |Y| * D_{\max}$ inequalities. If $x_i = d_j$, $c_{x_i,d_j} = 1$; otherwise $c_{x_i,d_j} = 0$. If $y_{d_j} = h$, $c_{y_{d_j},h} = 1$; otherwise $c_{y_{d_j},h} = 0$. By Theorem 4.12, SOFT_EGCC^{var} cost function is a PLPS cost function. □

Example 4.35. *Consider the following WCSP* $P = \{\mathcal{X}, \mathcal{D}, W_S, k\}$*:* $\mathcal{X} = \{x_1, x_2, y_a, y_b\}$, $D(x_1) = D(x_2) = \{a, b\}$, $D(y_a)D = (y_b) = \{0, 1, 2\}$, $W_S =$ SOFT_EGCC$^{var}(x_1, x_2, y_a, y_b)$. *For example,* $W_S(a, a, 2, 1) = 1$ *because* $|y_a - occ(a, (x_1, x_2))| + |y_b - occ(b, (x_1, x_2))| = 1$.

The corresponding integer linear program is:

$$\min L_1 + U_1 + L_2 + U_2 \;\; s.t.$$

$$c_{x_1,a} + c_{x_2,a} - c_{y_a,1} - 2c_{y_a,2} - L_1 + U_1 = 0$$

$$c_{x_1,b} + c_{x_2,b} - c_{y_b,1} - 2c_{y_b,2} - L_2 + U_2 = 0$$

$$L_1 \geq 0, U_1 \geq 0, L_2 \geq 0, U_2 \geq 0$$

$$c_{x_1,a} + c_{x_1,b} = 1$$

$$c_{x_2,a} + c_{x_2,b} = 1$$

$$c_{y_a,0} + c_{y_a,1} + c_{y_a,2} = 1$$

$$c_{y_b,0} + c_{y_b,1} + c_{y_b,2} = 1$$

where $c_{x_1,a}, c_{x_1,b}, c_{x_2,a}, c_{x_2,b}, c_{y_a,0}, c_{y_a,1}, c_{y_a,2}, c_{y_b,0}, c_{y_b,1}, c_{y_b,2} \in \{0, 1\}$.

4.5.3 The SOFT_DISJUNCTIVE/CUMULATIVE Cost Function

The DISJUNCTIVE$(S, p_1, \ldots, p_n)$ constraint [20] is used in non-preemptive scheduling. A set of n variables $S = x_1, \ldots, x_n$ is used to represent the beginning time of n jobs. Each job $x_i \in S$ has its process time p_i and its possible start time defined by its domain $d \in D(x_i)$. After a job has started, it cannot be interrupted to process another job. The DISJUNCTIVE constraint restricts that no more than one job can be processed at the same time. The CUMULATIVE($S, p_1, \ldots, p_n, k$) constraint allows k jobs to be processed at the same time instead of 1 in the DISJUNCTIVE constraint. We first define a set T which consists of every possible time in a constraint such that $T = \bigcup_{\forall x_i \in S} \{d + q \mid d \in D(x_i), 0 \leq q \leq p_i\}$.

Definition 4.36. *The* DISJUNCTIVE*($S, p_1, \ldots, p_n$) constraint holds if* $(x_i + p_i \leq x_j) \vee (x_j + p_j \leq x_i)$ *for every pair of* $x_i, x_j \in S$ *[20].*

Definition 4.37. *The* CUMULATIVE*($S, p_1, \ldots, p_n, k$) constraint holds if* $\forall t \in T$, $|\{x_i \mid x_i \leq t \leq x_i + p_i\}| \leq k$ *[20].*

Theorem 4.38. *Enforcing GAC on* DISJUNCTIVE *and* CUMULATIVE *constraints is NP-hard [1].*

The DISJUNCTIVE constraint is softened by allowing more than one job to be processed at the same time with a cost given as the penalty. The CUMULATIVE can also be softened by allowing more than k jobs can be processed at the same time with a penalty.

Definition 4.39. *Given the* DISJUNCTIVE*() constraint and an assignment tuple* ℓ *in variables* S*,*

$$\text{SOFT\_DISJUNCTIVE}^{val}(x_i, \ldots, x_n, p_1, \ldots, p_n)(\ell)$$

$$= \sum_{t=0}^{T} \sum_{i=1}^{n} \max(|\{i|\ell[x_i] \leq t \leq \ell[x_i] + p_i\}| - 1, 0)$$

Definition 4.40. *Given the* CUMULATIVE*() constraint and an assignment tuple* ℓ *in variables* S*,*

$$\text{SOFT\_CUMULATIVE}^{val}(x_i, \ldots, x_n, p_1, \ldots, p_n, k)(\ell)$$

$$= \sum_{t=0}^{T} \sum_{i=1}^{n} \max(|\{i|\ell[x_i] \leq t \leq \ell[x_i] + p_i\}| - k, 0)$$

Theorem 4.41. *Computing the minimum costs of* SOFT_DISJUNCTIVE^{val} *and* SOFT_CUMULATIVE^{val} *cost functions is NP-hard.*

Proof. As the SOFT_DISJUNCTIVE^{val} cost function is derived from the DISJUNCTIVE constraint and it is NP-hard to enforce GAC on the DISJUNCTIVE constraint. By Lemma 4.2, computing the minimum cost of SOFT_DISJUNCTIVE^{val} is NP-hard. The SOFT_CUMULATIVE^{val} cost function is a generalized version

of the SOFT_DISJUNCTIVE^{val} cost function, so computing the minimum cost of SOFT_DISJUNCTIVE^{val} is also NP-hard.

□

We can model this cost function in the form of a PLPS cost function such that we can compute the approximated minimum cost efficiently by linear relaxation.

Theorem 4.42. *The* SOFT_DISJUNCTIVE^{val} *and* SOFT_CUMULATIVE^{val} *cost functions are PLPS cost functions.*

Proof. The SOFT_DISJUNCTIVE^{val} cost function can be expressed as an integer linear program I where I is defined as:

$$\begin{array}{ll} \min \sum_{t \in T} U_t & \text{s.t.} \\ \sum_{i=1}^{n} \sum_{j=\max(t-p_i,0)}^{t} c_{x_i,j} - U_t \leq 1 & \forall t \in T \\ \sum_{d \in D(x_i)} c_{x_i,d} = 1 & \forall i = 1, 2, \ldots, n \end{array}$$

where $c_{x_i,d} \in \{0,1\}$ for all $x_i \in S$ and $d \in D(x_i)$. Let $D_{\max}$ be the maximum domain size for the variables in S, the corresponding integer linear program uses $|S|*D_{\max}+|T|$ variables and $|T|+|S|+|S|*D_{\max}$ inequalities. If $x_i = d$, $c_{x_i,d} = 1$; otherwise $c_{x_i,d} = 0$. By Theorem 4.12, the SOFT_DISJUNCTIVE^{val} cost function is a PLPS cost function.

The SOFT_CUMULATIVE^{val} cost function can be expressed as an integer linear program I where I is defined as:

$$\begin{array}{ll} \min \sum_{t \in T} U_t & \text{s.t.} \\ \sum_{i=1}^{n} \sum_{j=\max(t-p_i,0)}^{t} c_{x_i,j} - U_t \leq k & \forall t \in T \\ \sum_{d \in D(x_i)} c_{x_i,d} = 1 & \forall i = 1, 2, \ldots, n \end{array}$$

where $c_{x_i,d} \in \{0,1\}$ for all $x_i \in S$ and $d \in D(x_i)$. The corresponding integer linear program uses $|S|*D_{\max}+|T|$ variables and $|T|+|S|+|S|*D_{\max}$ inequalities. If $x_i = d$, $c_{x_i,d} = 1$; otherwise $c_{x_i,d} = 0$. By Theorem 4.12, the SOFT_DISJUNCTIVE^{val} cost function is a PLPS cost function. □

Example 4.43. *Consider the following WCSP* $P = \{\mathcal{X}, \mathcal{D}, W_S, k\}$*:* $\mathcal{X} = \{x_1, x_2\}$*,* $D(x_1) = \{0, 1, 2, 3\}$*,* $W_S = \text{SOFT\_DISJUNCTIVE}^{val}(x_1, x_2, 2, 3)$*. For example,* $W_S(2, 0) = 1$ *because when* $t = 2$*,* $x_1 \leq t \leq x_1 + 2$ *and* $x_2 \leq t \leq x_2 + 3$ *and the two jobs overlap each other, and* $\sum_{t=0}^{T} \sum_{i=1}^{n} |\{i | x_i \leq 2 \leq x_i + p_i\}| = 1$*. The corresponding integer linear program is:*

$$\min U_0 + U_1 + U_2 + U_3 + U_4 \;\; s.t.$$

$$c_{x_1,0} + c_{x_2,0} - U_0 \leq 0$$

$$c_{x_1,0} + c_{x_1,1} + c_{x_2,0} + c_{x_2,1} - U_1 \leq 0$$

$$c_{x_1,1} + c_{x_1,2} + c_{x_2,0} + c_{x_2,1} + c_{x_2,2} - U_2 \leq 0$$

$$c_{x_1,2} + c_{x_1,3} + c_{x_2,1} + c_{x_2,2} + c_{x_2,3} - U_3 \leq 0$$

$$c_{x_1,3} + c_{x_1,4} + c_{x_2,2} + c_{x_2,3} + c_{x_2,4} - U_4 \leq 0$$

$$U_0 \geq 0, U_1 \geq 0, U_2 \geq 0, U_3 \geq 0, U_4 \geq 0$$

$$c_{x_1,0} + c_{x_1,1} + c_{x_1,2} + c_{x_1,3} = 1$$

$$c_{x_2,0} + c_{x_2,1} + c_{x_2,2} + c_{x_2,3} = 1$$

where $c_{x_1,0}, c_{x_1,1}, c_{x_1,2}, c_{x_1,3}, c_{x_2,0}, c_{x_2,1}, c_{x_2,2}, c_{x_2,3} \in \{0, 1\}$.

4.6 Implementation Issues

In this section, we discuss the issues when we implement our framework into a WCSP solver. In our experiments, we use IBM ILOG CPLEX Optimizer 12.2 to solve the corresponding linear programs of PLPS cost functions. We discuss three main issues in our implementaion: (1) reducing the number of calls to linear programming solver for PLPS cost functions; (2) speeding up the linear programming solver by solving linear programs incrementally, and; (3) the floating point rounding

problem in the linear programming solver.

First, although enforcing relaxed consistencies on PLPS cost functions requires only polynomial time, it is still expensive to solve the linear programs. To reduce the number of calling the linear program solver, we include a data structure to remember the unbroken supports. To compute the cost of a value, we first check if the cost is affected by previous modifications. If it is the case, the cost is recomputed; otherwise the stored value is returned in order to save time.

Second, CPLEX can solve linear programs incrementally based on the solution of a similar linear program. Since enforcing consistencies on PLPS cost functions requires solving linear programs with minor modifications, we allow CPLEX to handle compute the solutions of linear programs incrementally instead of creating a new linear program whenever the domains and costs are modified. The same method can also be applied when a value is removed, which can be done by setting the upper bound of the corresponding value to 0.

Third, when integers are stored with floating point representation, an inevitable tiny error is often introduced and this case also happens in CPLEX and a bigger error will be introduced if the ceiling function is applied afterward. For example, if the minimum cost of a variable is 1 and a tiny error is added, applying the ceiling function on this variable returns 2 and a wrong value can be projected in enforcing relaxed consistencies. In order to avoid such error, we truncate the floating point numbers used in CPLEX at the 10th decimal place and minus a tiny number before finding the ceiling of these numbers.

4.7 Experimental Results

In this section, we first conduct experiments on the PLPS cost functions we have introduced, including the SOFT_SLIDINGSUM^{dec}, SOFT_EGCC^{var}, and SOFT_DISJUNCTIVE^{val} cost functions and demonstrate the efficiency of our framework. Finally we will discuss the results.

The benchmarks we use consist of hard constraints in nature and can be modeled as hard CSPs directly. We soften them by assigning a random unary cost from 0 to 9 to each value in the domain of each variable representing their preferences, and replacing the hard constraints with their soft variants.

To demonstrate the efficiency of PLPS cost functions and the use of their conjunctions, we compare the performances of the following models in this experiment. We include (a) models using PLPS cost functions, (b) models using conjoined PLPS cost functions, and (c) models using flow-based projection-safe cost functions. Since the SOFT_SLIDINGSUM^{dec} and SOFT_EGCC^{var} cost functions cannot be modeled directly as flow-based projection-safe cost functions, we model the instances with flow-based projection-safe cost functions by decomposing the SOFT_SLIDINGSUM^{dec} and SOFT_EGCC^{var} cost functions in the model (c). We also add (d) models using PLPS cost functions with decomposed SOFT_SLIDINGSUM^{dec} and SOFT_EGCC^{var} cost functions to compare with the model (c).

Since there is no well-known efficient or effective algorithm to model the SOFT_DISJUNCTIVE^{val} cost functions, model (c) and model (d) cannot be constructed. Instead, we compare (c) the model with the integer linear program approach without linear relaxation and show that the speedup by using the approximation by linear relaxation can compensate the enlarged search space of enforcing weaker consistencies.

The consistencies GAC*, FDGAC*, weak EDGAC* and their relaxed versions are implemented in Toulbar2 v0.9. IBM ILOG CPLEX Optimizer 12.2 is called from Toulbar2 to solve (integer) linear programs. Variables with smaller domains and values with lower unary costs are assigned first. The experiments are conducted on an Intel Core2 Duo E7400 (2 x 2.80GHz) machine with 4GB RAM. In each benchmark we use different parameter settings to construct different instances, and 10 random cases are generated with each parameter setting. We use the timeout of 3600 seconds and report the average number of backtracks (bt) and the average runtime in seconds (time) for solved cases. The runtime includes the CPU time used

by both the WCSP solver Toulbar2 and the linear program solver CPLEX. Next to the total CPU time, we also report separately in brackets the CPU time used by the linear program solver denoted as CPLEX (CPLEX). We truncate the floating point variables in CPLEX at the 10-th decimal place. We mark the entries with a "*" if the execution of one of the 10 instances exceeds the timeout. The best result among those with the most cases solved is highlighted in bold.

4.7.1 Generalized Car Sequencing Problem

The Generalized Car Sequencing Problem (Generalizing prob001 in CSPLib) is to find a sequence for n cars of $u \in U$ different types to be built. There is a set of options I which may or may not be equipped by each type, and each assembly line of an option $i \in I$ restricts that at most m_i cars for every s_i cars with that option equipped can be built. We generalize the problem such that a cost $c_{u,i}$ is required for each type of car $u \in U$ for each option $i \in I$ to be equipped, and each assembly line of an option $i \in I$ allows a maximum of m_i costs to be spent on that option for every s_i cars in total. A GCC constraint is used to ensure that the number of cars of each type is built according to the plan. The SOFT_SLIDINGSUM^{dec} cost functions are used to ensure the restrictions of each assembly line are satisfied. We fix $|I| = 3$ and $u = 5$ and use instances with different n in our experiments.

To model the problem with flow-based projection-safe cost functions, we decompose the SOFT_SLIDINGSUM^{dec} cost functions into SOFT_SUM^{val} cost functions, which can be modeled by the SOFT_REGULAR^{var} cost functions [30] as flow-based projection-safe cost functions.

Definition 4.44. *Given a tuple ℓ, the* SUM(S, l, u) *constraint holds if $l \leq \sum_{x_i \in S} \ell[x_i] \leq u$, where $\ell[x_i]$ is the value assigned to x_i in the tuple ℓ.*

Definition 4.45. *Given the* SUM*() constraint and an assignment tuple ℓ in variables S,*

$$\text{SOFT\_SUM}^{\text{val}}(S, l, u)(\ell) = \max(\sum_{x_i \in S} \ell[x_i] - u, l - \sum_{x_i \in S} \ell[x_i], 0)$$

Results are shown in Table 4.1. In the models with decomposed SOFT_SLIDINGSUM^{dec} cost functions (models (c) and (d)), model (d) requires more time than model (c) as the overhead of finding the minimum cost of a single PLPS cost function is greater than that of a flow-based projection-safe cost function. However, in model (a) and model (b), using PLPS cost functions without decomposing the SOFT_SLIDINGSUM^{dec} cost functions outperforms model (c). By conjoining PLPS cost functions, model (b) prunes even more and requires less time than other models. Since the instances only contain PLPS cost functions, they are conjoined into a single PLPS cost function in our model. As there is no possible propagation between cost functions, the effects of relaxed GAC*, relaxed FDGAC*, and relaxed weak EDGAC* are similar. So relaxed FDGAC* and relaxed weak EDGAC* do not infer a much better bound than relaxed GAC* when conjoined PLPS cost functions are used. The reduction in search space does not compensate for the pruning overhead, and the simpler and less costly relaxed GAC* gives the best results in terms of run-time.

4.7.2 Magic Series Problem

The Magic Series Problem (prob019 in CSPLib) is to find a sequence of n variables which forms a magic series. A non-empty finite series $S = (s_0, s_1, \ldots, s_n)$ is *m*agic if and only if there are s_i occurrences of $i \in S$ for each integer i ranging from 0 to n. For example, $S = (3, 2, 1, 1, 0, 0, 0)$ is an example of a magic series for $n = 6$ as there are three 0's, two 1's, a 2, a 3, and no 4, 5, and 6 in the series S. We use the SOFT_EGCC^{var} cost functions to restrict the occurrences of each values.

To model the problem with flow-based projection-safe cost functions, we decompose the SOFT_EGCC^{var} cost functions into SOFT_AMONG_VAR^{var} cost functions, which is a generalization of the SOFT_AMONG cost function with a count variable. Similar to SOFT_AMONG, The SOFT_AMONG_VAR^{var} cost function can be modeled as flow-based projection-safe cost function.

(a) Modeling with PLPS cost functions						
n	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
8	19.0	0.21 (0.20)	9.2	0.20 (0.19)	**9.0**	0.25 (0.23)
10	41.2	0.55 (0.51)	21.0	0.52 (0.49)	19.8	0.73 (0.68)
12	119.6	1.44 (1.35)	48.2	1.15 (1.07)	45.6	1.34 (1.26)
14	585.1	17.63 (16.91)	264.8	13.12 (12.76)	249.0	15.19 (14.61)
(b) Modeling with conjoined PLPS cost functions						
n	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
8	16.0	**0.19 (0.18)**	12.4	0.30 (0.28)	12.4	0.38 (0.35)
10	30.6	**0.46 (0.43)**	20.0	0.71 (0.65)	**17.8**	0.86 (0.80)
12	86.4	**1.07 (1.01)**	43.0	1.52 (1.45)	**36.4**	1.62 (1.55)
14	133.0	**1.30 (1.26)**	74.2	1.77 (1.71)	**64.1**	1.77 (1.72)
(c) Modeling with flow-based cost functions (SOFT_REGULAR)						
n	GAC*		FDGAC*		weak EDGAC*	
	bt	time	bt	time	bt	time
8	558.6	0.97	243.1	0.41	198.0	0.45
10	4023.4	1.57	865.2	0.72	559.1	0.68
12	55866.2	24.73	24496.9	22.15	6741.8	15.49
14	279748	152.24	104588	108.94	20341	65.13
(d) Modeling with PLPS cost functions (decomposed SOFT_SLIDINGSUM^{dec})						
n	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
8	558.6	23.13 (22.36)	243.1	9.53 (9.17)	198.0	14.08 (13.62)
10	4023.4	39.61 (39.07)	865.2	18.91 (18.12)	559.1	23.35 (22.86)
12	55866.2	730.32 (721.96)	24496.9	224.81 (219.15)	6741.8	276.92 (271.07)
14	*	*	*	*	*	*

Table 4.1: The generalized car sequencing problem using SOFT_SLIDINGSUM^{dec}

Definition 4.46. *The* AMONG_VAR(S, y, v) *constraint holds if* $y = occ(v, S)$*, where* $occ(v, S)$ *is the number of occurrences of* v *in* S*.*

Definition 4.47. *Given the* AMONG_VAR *constraint and an assignment tuple* ℓ *in variables* S*,*

$$\text{SOFT\_AMONG\_VAR}^{var}(S, y, v)(\ell) = |y - occ(v, S)|$$

Results are shown in Table 4.2. Similar to the last experiment, model (d) requires more time than model (c) as the overhead of PLPS cost functions are greater. On the other hand model (a) and model (b) outperform model (c). By conjoining PLPS cost functions, model (b) prunes more and requires less time than other

(a) Modeling with PLPS cost functions						
n	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
9	23.1	**0.24 (0.23)**	19.3	0.27 (0.26)	17.1	0.43 (0.41)
12	54.7	**0.71 (0.65)**	44.9	0.99 (0.93)	42.3	1.52 (1.43)
15	89.2	1.70 (1.59)	53.1	2.32 (2.21)	50.2	3.64 (3.46)
18	93.7	3.03 (2.89)	64.7	4.80 (4.64)	59.8	6.41 (6.13)
(b) Modeling with conjoined PLPS cost functions						
n	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
9	12.8	0.27 (0.25)	9.8	0.43 (0.40)	**9.7**	0.54 (0.50)
12	33.5	0.86 (0.81)	24.6	1.55 (1.48)	**24.3**	1.89 (1.80)
15	39.2	**1.62 (1.52)**	32.6	2.95 (2.86)	**32.3**	3.64 (3.51)
18	49.4	**2.96 (2.87)**	36.7	5.78 (5.48)	**36.4**	7.29 (6.82)
(c) Modeling with flow-based cost functions (SOFT_AMONG^{var})						
n	GAC*		FDGAC*		weak EDGAC*	
	bt	time	bt	time	bt	time
9	680.2	5.00	83.4	1.26	62.1	1.30
12	6141.8	220.22	252.3	19.15	213.4	18.82
15	*	*	809.9	228.03	539.2	203.14
18	*	*	*	*	*	*
(d) Modeling with PLPS cost functions (decomposed SOFT_EGCC^{var})						
n	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
9	680.2	164.12 (162.03)	83.4	29.76 (28.51)	62.1	38.41 (36.93)
12	*	*	252.3	533.12 (526.19)	213.4	679.03 (671.56)
15	*	*	*	*	*	*
18	*	*	*	*	*	*

Table 4.2: The magic square problem using SOFT_EGCC^{var}

models. (Relaxed) weak EDGAC* also prunes more than than (relaxed) FDGAC* and (relaxed) GAC* in all models with either PLPS cost functions or flow-based projection-safe cost functions. Similar to that of the last experiment, relaxed GAC* gives the best results in model (b) in terms of run-time as it is simpler and less costly.

4.7.3 Weighted Tardiness Scheduling Problem

The Weighted Tardiness Scheduling Problem (in OR-Library) is to find a schedule of n jobs to be processed, where no two jobs are processed at the same time. In each problem, there is n jobs and a set of total available time slots T. Each job

is given a time slot, if a job cannot be processed within the given time slot, a earliness/tardiness penalty is given. A SOFT_DISJUNCTIVE^{val} cost function is used to ensure no two jobs are processed at the same time. The unary costs are used to model the earliness/tardiness penalty of each job. In each instance, we use the number of jobs n, the size of the the total available time slots $|T|$, and the average duration of each job d as the parameters. A time slot with the length of $|T|/2$ is given to each job, and a random earliness/tardiness penalty is given to each job if it cannot be processed within the given time slot.

Since there is no other efficient way to model SOFT_DISJUNCTIVE^{val} cost functions in WCSP. Instead of the model (c) and (d) defined above, we compare the result of the linear cost function approach of SOFT_DISJUNCTIVE^{val} with the integer programming approach of the same implementation as model (c), which allows the exact minimum costs to be found and the common consistency algorithms in WCSP like GAC*, FDGAC*, *etc.*, to be enforced.

Results are shown in Table 4.3. In this benchmark, the integer linear program approach (model (c)) prunes more than modeling with PLPS cost functions (model (a)) as relaxed consistencies are weaker than standard consistencies. However it also takes much more time to solve and the extra pruning power offered in using integer linear programs does not pay off. By conjoining PLPS cost functions, model (b) prunes more and requires less time than other models. (Relaxed) weak EDGAC* also prunes more than than (relaxed) FDGAC* and (relaxed) GAC* in all models with either PLPS cost functions or flow-based projection-safe cost functions. Similar to that of the above experiments, relaxed GAC* gives the best results in model (b) in terms of run-time as it is simpler and less costly.

(a) Modeling with PLPS cost functions						
$n, d, \|T\|$	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
3,3,12	7.0	**0.05 (0.05)**	**6.0**	0.06 (0.06)	**6.0**	0.08 (0.08)
4,4,20	13.0	**0.14 (0.13)**	**8.0**	0.18 (0.17)	**8.0**	0.25 (0.24)
5,5,30	35.0	0.60 (0.56)	19.0	0.68 (0.63)	**15.0**	0.98 (0.92)
6,5,35	382.0	7.01 (6.75)	32.1	1.90 (1.82)	28.1	2.41 (2.32)
7,5,40	2253.6	61.89 (60.14)	27.0	2.78 (2.47)	25.2	3.51 (3.32)
8,5,45	*	* (*)	214.0	22.09 (21.23)	210.1	30.16 (28.90)
(b) Modeling with conjoined PLPS cost function						
$n, d, \|T\|$	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
3,3,12	**6.0**	**0.05 (0.05)**	**6.0**	0.09 (0.09)	**6.0**	0.16 (0.13)
4,4,20	**8.0**	0.15 (0.14)	**8.0**	0.29 (0.28)	**8.0**	0.35 (0.34)
5,5,30	15.5	**0.53 (0.50)**	15.2	1.04 (1.01)	**15.0**	1.29 (1.25)
6,5,35	23.2	**0.95 (0.90)**	18.8	1.90 (1.83)	**18.0**	2.32 (2.24)
7,5,40	35.2	**1.72 (1.64)**	27.0	3.44 (3.30)	**21.0**	4.23 (4.07)
8,5,45	40.1	**3.49 (3.24)**	34.5	7.38 (7.08)	**33.1**	8.71 (8.31)
(c) Modeling with PLPS cost functions, linear programs solved as integer programs						
$n, d, \|T\|$	GAC*		FDGAC*		weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
3,3,12	6.8	1.02 (0.96)	**6.0**	1.37 (1.29)	**6.0**	1.84 (1.78)
4,4,20	12.7	6.62 (6.34)	**8.0**	7.56 (7.23)	**8.0**	9.37 (8.93)
5,5,30	34.8	38.28 (37.71)	**15.0**	40.77 (40.20)	**15.0**	54.12 (53.69)
6,5,35	61.1	100.99 (99.82)	19.0	121.82 (120.13)	**18.0**	153.09 (151.62)
7,5,40	81.0	219.85 (213.13)	23.2	302.98 (292.65)	22.8	453.10 (440.12)
8,5,45	*	*	*	*	*	*

Table 4.3: The weighted tardiness scheduling problem using SOFT_DISJUNCTIVE^{val}

Chapter 5

Integral Polynomially Linear Projection-Safe Cost Functions

In this chapter, we propose *Integral Polynomially Linear Projection-Safe (IPLPS)* cost functions as a subclass of PLPS cost functions. Solving the corresponding integer linear programs of IPLPS cost functions with linear relaxation always gives integral minimums. Given a standard WCSP consistency α, we give theorems showing that maintaining a relaxed consistencies α on a conjunction of IPLPS cost functions is strictly stronger than maintaining α on the individual cost functions. A useful application of our method is on some IPLPS global cost functions, whose minimum cost computations are tractable and yet those for their conjunctions are not. We show that flow-based projection-safe and polynomially decomposable cost functions fall into this category. Experiments are conducted to confirm empirically that performing relaxed consistencies on the conjoined cost functions is more efficient than performing the corresponding standard consistencies on the individual cost functions.

5.1 Integral Polynomially Linear Projection-Safe Cost Functions

Integral polynomially linear projection-safe (IPLPS) cost functions form a special subclass of PLPS cost functions. A cost function W_S is *integral polynomially linear* if (a) W_S is linear, (b) the size of the corresponding integer program is polynomial to the number of variables and the maximum domain size, and (c) the optimal solution, if it exists, of the linear relaxation of its corresponding linear integer program I_{W_S} is always integral.

Lemma 5.1. *Integral polynomially linear cost functions are polynomially linear.*

An immediate observation is that the exact minimum cost of an integral linear cost function can be obtained by solving the linear relaxation of their corresponding integer linear programs.

Lemma 5.2. *If W_S is an integral polynomially linear cost function,* $\min\{W_S\} = \text{approx\_min}\{W_S\}$.

Theorem 5.3. *Minimum cost computation of integral polynomially linear cost functions is polynomial.*

Proof. Since $\min\{W_S\} = \text{approx\_min}\{W_S\}$, $\min\{W_S\}$ can be determined using interior point algorithms [49] for linear programs with the worst case complexity bounded by polynomial time. □

Recall the notion of $\mathcal{T}$ projection-safety. In addition to flow-basedness and polynomially linearity, integral polynomially linearity is another good property $\mathcal{T}$ to be maintained across projections/extensions. Therefore, it makes sense to require cost functions to be *integral polynomially linear projection-safe (IPLPS)*.

We give the possible sufficient conditions to identify IPLPS cost functions.

Theorem 5.4. *A cost function W_S is* integral polynomially linear projection-safe *if:*

1. *W_S is PLPS and has the corresponding integer linear program I_{W_S}, and;*

2. *I_{W_S} is* totally dual integral *or the associated matrix of I_{W_S} is* totally unimodular.

Proof. By lemma 4.11, PLPS cost functions remain PLPS after projections and extensions, so W_S is PLPS after projections and extensions given the condition 1). In addition, if a linear program is totally dual integral or its associated matrix is totally unimodular, its optimal solutions must be integral [36]. Since projections and extensions can be performed on W_S by adding terms to the objective function of I_{W_S}. The structure of I_{W_S} remains unchanged and the condition 2) is preserved after projections and extensions.

As a result, we can construct the sufficient conditions for IPLPS cost functions as above. □

Integral polynomially linear and polynomially linear projection-safe cost functions are interesting since their conjunctions are PLPS.

By Lemma 4.27 and 5.1, we have the following corollary.

Corollary 5.5. *Suppose W_{S_1} and W_{S_2} are IPLPS cost functions. The conjunction $W_{conj} \equiv W_{S_1} \wedge W_{S_2}$ is PLPS.*

Corollary 5.6. *Suppose W_S is IPLPS, and α-consistency is one of GAC*, FDGAC* and weak EDGAC*. Relaxed α-consistent on W_S is equivalent to α-consistent on W_S.*

In general, it is NP-hard to compute the minimum cost of the conjunction of overlapping IPLPS cost functions. On the other hand, the conjunction of their corresponding linear programs may not always give integral minimums when there exists a minimum [49]. As the conjunction of IPLPS cost functions remains PLPS, linear programming techniques allow its approximated minimum cost to be computed efficiently, and relaxed form of standard consistencies can thus be enforced. We have

the following result when relaxed consistencies are enforced on the conjunction of IPLPS cost functions compared to the corresponding (non-relaxed) consistencies enforced on the individual cost functions.

Given WCSP $P_{IPLPS} = (\mathcal{X}, \mathcal{D}, \mathcal{C}_{IPLPS}, k)$, where each cost function $W_S \in \mathcal{C}_{IPLPS}$ is IPLPS with corresponding integer linear program I_{W_S}. We assume that $\mathcal{C}_{IPLPS}$ contains overlapping cost functions. We can construct an equivalent WCSP $P_{conj} = (\mathcal{X}, \mathcal{D}, \mathcal{C}_{conj}, k)$ where $\mathcal{C}_{conj} = \{W_{conj}\}$ and $W_{conj} \equiv \bigwedge_{W_S \in \mathcal{C}_{IPLPS}} W_S$ with the corresponding scope $S_{conj} \equiv \bigcup_{W_S \in \mathcal{C}_{IPLPS}} S$ and integer linear program $I_{W_{conj}} \equiv \bigwedge_{W_S \in \mathcal{C}_{IPLPS}} I_{W_S}$.

We show that relaxed (FD)GAC* and relaxed weak EDGAC* on P_{conj} are strictly stronger than (FD)GAC* and weak EDGAC* on P_{IPLPS} respectively by the following theorem.

Theorem 5.7. *Suppose α-consistency is one of GAC*, FDGAC* and weak EDGAC*. Relaxed α-consistent on P_{conj} is strictly stronger than α-consistent on P_{IPLPS}.*

Proof. We prove the part for relaxed GAC*. The proofs for the other consistencies are similar.

Assume P_{conj} is relaxed GAC*, but P_{IPLPS} is not GAC*. There exists a variable $x_i \in \mathcal{X}$ with a value $a \in D(x_i)$ and a cost function $W_S \in \mathcal{C}_{IPLPS}$ in P_{IPLPS} such that $\min\{W_S(\ell) \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S)\} > 0$. Since all cost functions $W_S \in \mathcal{C}_{IPLPS}$ are IPLPS, we have

$$\begin{aligned} & \text{approx\_min}\{W_{conj} \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S)\} \\ \geq\ & \bigoplus_{W_S \in \mathcal{C}_{IPLPS}} \text{approx\_min}\{W_S \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S)\} \\ =\ & \bigoplus_{W_S \in \mathcal{C}_{IPLPS}} \min\{W_S(\ell) \mid \ell[x_i] = a \wedge \ell \in \mathcal{L}(S)\} > 0 \end{aligned}$$

Thus, a cannot have simple support and x_i cannot be relaxed GAC* with respect to W_{conj} in P_{conj}.

Consider $W_{S_1} = \text{SOFT\_ALLDIFF}^{var}(x_1, x_2, x_3)$ and $W_{S_2} = \text{SOFT\_ALLDIFF}^{var}(x_2, x_3, x_4)$, where $D(x_1) = \{a, b\}$, $D(x_2) = D(x_3) = \{a, b, c\}$

and $D(x_4) = \{b, c\}$. It is easy to check that $P_{IPLPS} = (\mathcal{X}, \mathcal{D}, \{W_{S_1}, W_{S_2}\}, k)$ is GAC*. However, $P_{conj} = (\mathcal{X}, \mathcal{D}, \{W_{S_1} \wedge W_{S_2}\}, k)$ is not relaxed GAC* since the approximated minimum cost when $x_1 = a$ is $1 > 0$.

Result follows. □

Since relaxed consistencies are the weaker forms of standard consistencies, we have the following lemma.

Lemma 5.8. *Suppose α-consistency is one of GAC*, FDGAC* and weak EDGAC*. We have α-consistent on P_{conj} > relaxed α-consistent on P_{conj}.*

Enforcing α-consistency on P_{conj} infers better bounds, but it can be NP-hard if computing the minimum costs of conjunctions of IPLPS cost functions is NP-hard. It may not be worthwhile to do so, while relaxed consistencies can still be enforced efficiently on P_{conj}.

5.2 Conjoining Global Cost Functions as IPLPS

An immediate application of Theorem 5.7 is to existing global cost functions with polytime minimum cost computation. In many cases the minimum cost computation for their conjunctions is NP-hard. Theorem 5.7 suggest that it is still worthwhile to enforce relaxed consistencies on these cost functions. Flow-based projection-safe cost functions [28, 30] and polynomially decomposable cost functions [31] are such examples. By enforcing relaxed consistencies on their conjunctions, the search benefits from the better bounds inferred.

Theorem 5.9. *Flow-based projection-safe cost functions are IPLPS.*

Proof. Every flow-based projection-safe cost function has a corresponding network flow problem, which in turn has a corresponding integer linear program with a totally unimodular matrix [38]. The cost function, the flow problem, and the integer linear program shares the same minimum cost. Since the integer linear program always has integral solutions when solved with linear relaxation, the result

follows. □

Corollary 5.10. *The flow-based projection-safe cost functions [30, 29]* SOFT_ALLDIFF^{var}, SOFT_ALLDIFF^{dec}, SOFT_GCC^{var}, SOFT_GCC^{val}, SOFT_SAME^{var}, SOFT_SAME^{val}, SOFT_REGULAR^{var}, *and* SOFT_REGULAR^{edit} *are IPLPS cost functions.*

Lee *et al.* [31] define polynomially decomposable cost functions and give their sufficient conditions. They further give some examples of polynomially decomposable cost functions with the corresponding distributive cost aggregation function. Those examples fulfill the sufficient conditions of polynomially decomposable cost functions as they are using the stated distributive aggregation function. Here we give the related definitions and show that those cost functions are also IPLPS cost functions.

Definition 5.11. *[31] A cost function W_S can be* safely decomposed *to a set of cost functions $\Omega = \{\omega_{S_1}, \ldots, \omega_{s_m}\}$ using cost aggregation function f, where $S_i \subseteq S$, iff*

1. $W_S(\ell) = f(\{\omega_{S_i}(\ell[S_i]) \mid \omega_{S_i} \in \Omega\})$*, and;*
2. f *is* distributive, i.e.
   (a) $\min\{W_{S_i}\} = f(\{\min\{\omega_{S_i}\} \mid \omega_{S_i} \in \Omega\})$*, and;*
   (b) *For a variable $x \in S$, a cost α and a tuple $\ell \in \mathcal{L}(S)$, $W_S(\ell) \oplus \alpha = f(\{\omega_{S_i}(\ell[S_i]) \oplus \nu_{x,S_i}(\alpha) \mid \omega_{S_i} \in \Omega\})$ and $W_S(\ell) \ominus \alpha = f(\{\omega_{S_i}(\ell[S_i]) \ominus \nu_{x,S_i}(\alpha) \mid \omega_{S_i} \in \Omega\})$, where the function ν is defined as $\nu_{x,S_i}(\alpha) = \alpha$ if $x \in S_i$, and 0 otherwise.*

A cost function W_S can be polynomially decomposed *into a set of cost functions $\Omega = \{\omega_{S_1}, \ldots, \omega_{s_m}\}$, where $S_i \subseteq S$, if*

1. *m is polynomial to the size of S and maximum domain size d,*

2. *Each* $\omega_{S_i} \in \Omega \cup \{\omega_{S_{m+1}}\}$*, where* $\omega_{S_{m+1}} = W_S$*, is either a tractable unary cost function, or can be safely decomposed into* $\Omega_i \subseteq \{\omega_{S_j} \mid j < i\}$ *using a tractable cost aggregation function* f_i*.*

Lemma 5.12. *[31] If a global cost function* W_S *can be represented as* $W_S(\ell) = \min_{i=1}^{r}\{\bigoplus_{j=1}^{n_i} \omega_{S_{i,j}}(\ell[S_{i,j}])\}$*, where:*

1. $\sum_{i=1}^{r} n_i$ *is polynomial to* $|S|$ *and* d*, and;*

2. *for each* i*,* $S_{i,j} \cap S_{i,k} = \varnothing$ *iff* $j \neq k$ *and* $\bigcup_{j}^{n_i} S_{i,j} = S$*,*

then W_S *is safely decomposable.*

Theorem 5.13. *Suppose* W_S *is a polynomially decomposable cost function using the aggregation function stated in Lemma 5.12, then* W_S *is IPLPS.*

Proof. We show that W_S is IPLPS by first showing that it is flow-based projection-safe. The aggregation function stated in Lemma 5.12 consists of the operations $\min$ and $\bigoplus$, which can be represented and computed in flow networks. So, W_S can be represented as a min-cost flow problem with a corresponding flow network, where each cost function $\omega_{S_{i,j}}$ is represented by a node. The operation $\min$ is represented by a new node as the sink and all the nodes of the related cost functions are connected to it. The operation $\bigoplus$ is represented by a path connecting all the nodes of the related cost functions.

As a result, W_S is a flow-based projection-safe cost function. By Theorem 5.9, W_S is an IPLPS cost functions. □

Corollary 5.14. *The polynomially decomposable cost functions [31]* SOFT_AMONG^{var}*,* SOFT_REGULAR^{var}*,* SOFT_GRAMMAR^{var}*,* MAX_WEIGHT*, and* MIN_WEIGHT *use the aggregation function stated in Lemma 5.12 and they are IPLPS.*

We note that, for the cost functions mentioned above, their dedicated polynomial time algorithms are usually more efficient than interior point algorithms or linear programming.

By propagating the conjunction of cost functions, extra pruning opportunities can be discovered which may reduce more search space than propagating the individual cost functions. Unfortunately, it can be NP-hard to compute the minimum cost for the conjunction of IPLPS cost functions even an efficient polynomial time algorithm is given for the individual cost functions.

Bessière *et al.* [8] show the above result on the hard ALLDIFF constraints and it can be generalized to the SOFT_ALLDIFF^{var}, SOFT_ALLDIFF^{dec}, SOFT_GCC^{var}, SOFT_GCC^{val}, and SOFT_SAME^{var} cost functions. Régin [42] also shows the above result on the hard AMONG [3] constraints, where an AMONG constraint restricts the number of variables to be assigned to a value from a specific set. The result can be generalized to the SOFT_AMONG^{var}, SOFT_REGULAR^{var}, SOFT_REGULAR^{edit}, and SOFT_GRAMMAR^{var} cost functions. Theorem 5.7 suggests that enforcing the relaxed consistencies on the conjunction of such IPLPS cost functions can still be more efficient and worthwhile than handling them individually.

Given WCSP $P_{IPLPS} = (\mathcal{X}, \mathcal{D}, \mathcal{C}_{IPLPS}, k)$, where $\mathcal{C}_{IPLPS}$ consists of some IPLPS cost functions, and an equivalent WCSP $P_{conj} = (\mathcal{X}, \mathcal{D}, \mathcal{C}_{conj}, k)$ where $\mathcal{C}_{conj} = \{W_{conj}\}$ and $W_{conj} \equiv \bigwedge_{W_S \in \mathcal{C}_{IPLPS}} W_S$. We give an example similar to the one given by Bessière *et al.* [8] in the following theorem. By propagating on a conjunction of IPLPS cost functions with relaxed consistencies, a higher bound can be inferred earlier in an exponentially number of steps during branch-and-bound search in such a case.

Theorem 5.15. *Suppose α-consistency is one of GAC*, FDGAC* and weak EDGAC*. There exists a class of WCSP P_{IPLPS}, so that if we enforce α-consistency on P_{conj} and α-consistency on P_{IPLPS} in branch-and-bound search, an exponential search tree needs to be explored for P_{IPLPS} to infer the same minimum cost as in the case of P_{conj}.*

Proof. We prove the part for relaxed GAC*. The proofs for the other consistencies are similar. Given a WCSP $P_{IPLPS} = (X \cup Y \cup Z, \mathcal{D}, \mathcal{C}_{IPLPS}, k)$

where $X = \{x_1, \ldots, x_n\}$, $Y = \{x_1, \ldots, x_{2n}\}$, $Z = \{x_1, \ldots, x_n\}$, $\mathcal{C}_{IPLPS} = \{\text{SOFT\_ALLDIFF}^{var}(X \cup Y), \text{SOFT\_ALLDIFF}^{var}(Y \cup Z)\}$, $D(X_i) = [1, 2n-1]$, $i = 1, \ldots, n$, $D(Y_i) = [1, 4n-1]$, $i = 1, \ldots, 2n$, and $D(Z_i) = [2n, 4n-1]$, $i = 1, \ldots, n$.

Consider the WCSP $P_{conj} = (X \cup Y \cup Z, \mathcal{D}, \mathcal{C}_{conj}, k)$ where $\mathcal{C}_{conj} = \{W_{conj}\}$ and $W_{conj} \equiv \text{SOFT\_ALLDIFF}^{var}(X \cup Y) \wedge \text{SOFT\_ALLDIFF}^{var}(Y \cup Z)$. W_{conj} gives an approximated minimum cost approx_$\min\{W_{conj}\}$ of 1 which can be inferred by enforcing relaxed GAC* on $\mathcal{C}_{conj}$. On the other hand, a subset of n or fewer variables has at least $2n-1$ values in their domains and a subset of $n+1$ to $3n$ variables has $4n-1$ values in their domains. Thus, to infer a minimum cost of 1 in P_{IPLPS} by enforcing GAC* on $\mathcal{C}_{IPLPS}$, we must instantiate at least $n-1$ variables. □

In addition to the theoretical results, we conduct experiments to show the efficiency of modeling cost functions as IPLPS cost functions and propagating their conjunctions in the next section.

5.3 Experimental Results

To demonstrate the efficiency of our framework, we compare the performances of (a) models using conjunctions of IPLPS cost functions against (b) models using individual flow-based projection-safe / polynomially decomposable cost functions. The consistencies GAC*, FDGAC*, weak EDGAC* and their relaxed versions are implemented in Toulbar2 v0.9. IBM ILOG CPLEX Optimizer 12.2 is called from Toulbar2 to solve (integer) linear programs. Our benchmarks' models consist of both IPLPS global cost functions as well as table cost functions, the latter of which are handled individually using exact minimum costs even when relaxed consistencies are used.

Variables with smaller domains and values with lower unary costs are assigned

first. The experiments are conducted on an Intel Core2 Duo E7400 (2 x 2.80GHz) machine with 4GB RAM. In each benchmark we use different parameter settings to construct different instances and 10 random cases are generated with each parameter setting. We use the timeout of 3600 seconds and report the average number of backtracks (bt) and the average runtime in seconds (time) for solved cases. The runtime includes the CPU time used by both the WCSP solver Toulbar2 and the linear programming solver CPLEX. Next to the runtime, we also report separately in brackets the CPU time used by CPLEX denoted as (CPLEX). We truncate the floating point variables in CPLEX at the 10-th decimal place. We mark the entries with a "*" if the execution of one of the 10 instances exceeds the timeout. The best result among those with the most cases solved is highlighted in bold.

To utilize the global cost functions described above, we soften the following problems by replacing the global constraints by their soft variants, by either the flow-based projection-safe / polynomially decomposable implementations or the IPLPS implementations. For each variable x_i introduced, a random unary cost from 0 to 9 is assigned to each value in $D(x_i)$. Random preferences are added to the instances in the form of table cost functions. Note that models using IPLPS cost functions contain also table cost functions and are thus applied with a mix of relaxed α-consistency (for IPLPS functions) and α-consistency (for table functions).

5.3.1 Car Sequencing Problem

The car sequencing problem (prob001 in CSPLib) finds a sequence of n cars of $u \in U$ different types to be built. There is a set of options I which may or may not be equipped by each type and each assembly line of an option $i \in I$ restricts that at most m_i cars for every s_i cars with that option equipped can be built. A GCC [41] constraint is used to ensure that the number of cars of each type is built according to the plan. Overlapping SOFT_AMONG$^{var}()$ [47] cost functions are used to ensure the restrictions of each assembly line are satisfied and they are modeled

Modeling with the conjunction of IPLPS cost functions						
n	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
12	72.6	2.22 (1.58)	12.1	**0.47 (0.33)**	**12.0**	0.85 (0.67)
14	85.7	3.11 (2.39)	15.8	**0.73 (0.55)**	**15.0**	1.30 (1.07)
16	89.3	4.33 (3.47)	16.1	**1.13 (0.87)**	**15.6**	1.92 (1.59)
18	123.3	7.20 (5.87)	18.9	**1.38 (1.06)**	**18.0**	2.24 (1.89)
20	139.7	10.29 (8.51)	22.0	**2.01 (1.49)**	**20.6**	3.31 (2.69)
Modeling with polynomially decomposable cost functions						
n	GAC*		FDGAC*		weak EDGAC*	
	bt	time	bt	time	bt	time
12	23667.9	23.03	563.4	2.67	210.3	1.54
14	310845	328.49	2774.9	16.53	983.1	11.89
16	*	*	6653.2	53.06	2191.3	25.10
18	*	*	8104.2	93.87	3651.7	49.62
20	*	*	21285.5	303.10	8025.6	161.82

Table 5.1: The soft car sequencing problem

by either polynomially decomposable cost functions or IPLPS cost functions. There are preferences for each assembly line, e.g. two consecutive cars of the same type are preferred, and they are modeled by table cost functions. We fix $|I| = 5$ and $u = 5$ and use instances with different n in our experiments.

Results are shown in Table 5.1. The model using conjunctions of IPLPS cost functions using relaxed α-consistency run faster and prune more than the model with individual flow-based projection-safe / polynomially decomposable cost functions using α-consistency in many cases, especially when the problem size is large.

As stronger consistencies have higher overhead, we gain in runtime only when the extra prunings can compensate for the overhead. This is not the case in general for relaxed weak EDGAC* in our easy problem instances as reported in these tables. That is why relaxed FDGAC* exhibits better runtime behavior than weak EDGAC*.

Modeling with the conjunction of IPLPS cost functions						
n, d	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
25, 8	5507.6	405.67 (379.50)	29.8	3.77 (3.25)	**25.4**	4.32 (3.66)
30, 8	*	*	63.4	18.37 (16.09)	**50.4**	8.50 (6.99)
35, 8	*	*	35.8	7.96 (5.62)	**35.0**	10.85 (7.21)
30, 12	*	*	140.1	**26.49 (20.64)**	**124.7**	30.88 (22.91)
35, 12	*	*	93.0	**45.02 (37.15)**	**78.3**	51.41 (40.31)
Modeling with flow-based projection-safe cost functions						
n, d	GAC*		FDGAC*		weak EDGAC*	
	bt	time	bt	time	bt	time
25, 8	16747.8	41.514	97.8	**0.67**	92.6	0.68
30, 8	*	*	224.0	**7.93**	208.4	8.75
35, 8	*	*	72.2	0.51	62.4	**0.44**
30, 12	*	*	*	*	*	*
35, 12	*	*	*	*	*	*

Table 5.2: The soft examination timetabling problem

5.3.2 Examination Timetabling Problem

The examination timetabling problem finds a schedule for n examinations over d days for s groups of students, Each group of students attends a set of at most d examinations and the number of days with more than 1 examination should be minimized for every group of students. A SOFT_ALLDIFF$^{var}()$ [40] cost function is used for every group of student, and they are modeled by either flow-based projection-safe cost functions or IPLPS cost functions. There are preferences between examinations, e.g. the locations of two examinations are far away and should not be scheduled on the same day in case there are students attending both of them and they are modeled by table cost functions. We fix $s = 4$ and use different n and d in our experiments.

Results are shown in Table 5.2. Similar to the last experiment, models using conjunctions of IPLPS cost functions using relaxed α-consistency run faster and prune more than models with individual flow-based projection-safe / polynomially decomposable cost functions using α-consistency in most cases. Also relaxed FDGAC* exhibits better runtime behavior than weak EDGAC* since the problem instances used are easy and the overhead of stronger consistency is not compensated by the

Modeling with the conjunction of IPLPS cost functions						
n	relaxed GAC*		relaxed FDGAC*		relaxed weak EDGAC*	
	bt	time (CPLEX)	bt	time (CPLEX)	bt	time (CPLEX)
6	2253.1	617.2	67.1	4.19 (4.01)	**45.1**	5.89 (5.71)
8	*	*	78.2	**6.02 (5.86)**	**54.1**	8.01 (7.79)
10	*	*	125.3	**10.27 (9.81)**	**79.3**	13.23 (12.77)
12	*	*	183.5	**21.50 (20.08)**	**98.4**	22.10 (20.46)
Modeling with flow-based projection-safe cost functions						
n	GAC*		FDGAC*		weak EDGAC*	
	bt	time	bt	time	bt	time
6	*	*	231.8	4.89	196.4	**3.56**
8	*	*	769.7	9.88	438.9	7.52
10	*	*	2031.4	103.52	802.3	65.17
12	*	*	*	*	*	*

Table 5.3: The soft fair scheduling problem

extra prunings.

5.3.3 Fair Scheduling

The fair scheduling problem [2] consists of n groups of people, each of them can be scheduled into one of s shifts over d days. Among each group of people and a specific period within the d days, the schedule should be *fair* such that they attend the same number of shift for every shift in s in that period. For example, given a problem with $n = 2$, $s = 4$, and $d = 4$, a fair schedule over all the 4 days is that both p_1 and p_2 are assigned to the shift 2 and shift 3 once, and the shift 2 twice. If p_1 is assigned to all of the shift 1, shift 2, shift 3, and shift 4 once instead, it is not a fair schedule. There are preferences between some groups. For example, there are groups preferred to be scheduled in the same shift. Such preferences are modeled by table cost functions. We model the problem by a set of variables $\{x_{ij}\}$ denoting the shift the i^{th} person is assigned to on the j^{th} day. We use the SOFT_SAME^{var} cost functions to model the restrictions. We fix $s = 5$ and $d = 5$ and use different n in our experiment.

Results are shown in Table 5.3. Similar to the last experiment, models using conjunctions of IPLPS cost functions using relaxed α-consistency run faster and prune

more than models with individual flow-based projection-safe / polynomially decomposable cost functions using α-consistency in most cases. Also relaxed FDGAC* exhibits better runtime behavior than weak EDGAC* since the problem instances used are easy and the overhead of stronger consistency is not compensated by the extra prunings.

5.3.4 Comparing WCSP Approach with Integer Linear programming Approach

We use slightly easier problem instances so that we can make sensible comparisons with the weaker consistencies and the flow-based projection-safe / polynomially decomposable cost function implementations. Note that integer linear programming solver can also solve our benchmarks competitively. We use more difficult instances with more preferences (table cost functions) to compare the performances of modeling the problem with *integer linear programs* (ILPs) solved by the IBM ILOG CPLEX Optimizer 12.2 with both of the models above. We use the encoding method introduced by Koster [23] to formulate binary cost functions as integer linear programs. We only show the results for the models with flow-based projection-safe / polynomially decomposable (p.d.) cost functions using weak EDGAC* and IPLPS cost functions using relaxed weak EDGAC* as those models have the best results among the other (relaxed) consistencies in the same model in this setting. Similar to the experiments we have conducted above, the model using IPLPS cost functions contains table cost functions and it is thus applied with a mix of relaxed weak EDGAC* (for IPLPS functions) and weak EDGAC* (for table functions).

n	p.d. & weak EDGAC*		IPLPS & relaxed weak EDGAC*		ILPs
	bt	time	bt	time (CPLEX)	time
12	527.8	119.96	**37.8**	103.26 (68.73)	**63.28**
14	2287.2	788.94	**42.6**	**155.21 (135.49)**	177.79
16	6835.1	1828.22	**96.3**	**207.07 (175.64)**	386.30
18	*	*	**110.1**	**653.82 (549.44)**	662.56
20	*	*	**311.2**	**1163.03 (1026.89)**	1442.44

Table 5.4: Comparison with integer linear programming: soft car sequencing

n, d	flow-based & weak EDGAC*		IPLPS & relaxed weak EDGAC*		ILPs
	bt	time	bt	time (CPLEX)	time
25, 8	211.0	2.93	**47.0**	5.87 (4.22)	**2.29**
30, 8	1140.1	31.28	**105.0**	11.53 (9.61)	**10.76**
35, 8	704.2	19.77	**84.1**	**11.07 (8.17)**	12.56
30, 12	*	*	**790.1**	**544.01 (449.89)**	725.54
35, 12	*	*	**681.0**	**738.09 (640.58)**	876.47

Table 5.5: Comparison with integer linear programming: soft examination timetabling

n, d	flow-based & weak EDGAC*		IPLPS & relaxed weak EDGAC*		ILPs
	bt	time	bt	time (CPLEX)	time
6	355.6	8.07	**53.8**	**7.18 (6.51)**	8.91
8	973.4	35.88	**155.4**	**31.82 (26.44)**	35.96
10	*	*	**413.0**	**286.13 (223.08)**	325.92
12	*	*	**892.3**	**923.21 (813.51)**	1315.61

Table 5.6: Comparison with integer linear programming: soft fair scheduling

Results are shown in Tables 5.4, 5.5, and 5.6. In almost all cases, our models using conjunctions of IPLPS cost functions run faster and prune more than the models with individual flow-based projection-safe / polynomially decomposable cost functions using α-consistency. On the other hand, our model runs faster in general when compared with the integer linear programming model using CPLEX as the integer linear program solver. The trend is more apparent when the problem size grows.

Chapter 6

Conclusions

In this chapter, we summarize the contributions of the thesis. We also propose possible future directions of our research.

6.1 Contributions

In this thesis, we enhance the weighted constraint satisfaction by introducing the concept of *polynomially linear projection-safe (PLPS) cost functions*. We define relaxed consistencies for polynomially linear projection-safe cost functions based on the existing standard consistencies. In addition, we demonstrate the benefits of conjoining such cost functions experimentally, and defined *integer polynomially linear projection-safe (IPLPS) cost functions* as a special subclass of PLPS cost functions to characterize the strength of the relaxed consistency notions on the conjunctions of IPLPS cost functions. Our contributions are five-fold.

First, we define the *polynomially linear projection-safe (PLPS) cost functions* based on their *integer linear program* formulations with size polynomial to their number of variables and maximum domain size. Their minimum costs can be computed by solving their related integer linear programs. We give the sufficient conditions for polynomially linear projection-safe cost functions whose properties are preserved in projections and extensions.

Second, we propose the relaxed consistencies on PLPS cost functions, which are

weaker but the enforcement can be much more efficient compared to the standard counterparts. The approximated minimum costs of PLPS cost functions can be computed by solving their related integer linear programs with linear relaxations. We give proofs for the feasibility of projecting the the smallest integral cost which is not less than the approximated minimum cost. Thus, we can define the relaxed version for the standard consistency notions including GAC*, FDGAC*, and weak EDGAC* by reformulating their requirements based on the minimum costs of a set of cost functions and replaced by their approximated minimum costs.

Third, we propose the use of the conjunctions of PLPS cost functions, which gives benefits in terms of pruning and runtime shown by experiments. We show that the conjunctions of PLPS cost functions remain PLPS, in which relaxed consistencies can still be applied on them. We show that propagating on a conjunction using the standard consistencies is stronger than propagating on the individual cost functions. Although it is not always true when relaxed consistencies are enforced, the benefits of using the conjunctions of PLPS cost functions are shown experimentally.

Fourth, we define *integral polynomially linear projection-safe (IPLPS) cost functions*, which is a subclass of PLPS cost functions and we characterize the strength of the relaxed consistency notions on the conjunctions of IPLPS cost functions over the strength of the corresponding standard consistency notions on the individual IPLPS cost functions. IPLPS cost functions are special PLPS cost functions and their exact minimum costs can be computed by solving their related integer linear programs with linear relaxation. In addition, the minimum cost of an IPLPS function can be computed in polynomial time. The same is not necessarily true for the conjunctions of IPLPS cost functions, which we show to be still PLPS. Our central results show that propagating on individual IPLPS cost functions using the standard (or relaxed since they are the same) consistencies is weaker than propagating on the conjunction of all these IPLPS cost functions using the relaxed versions of the consistencies, which is in turn weaker than propagating on the conjunction

using the standard consistency. The latter is NP-hard in general. Therefore, it is always more desirable to propagate on conjunctions of IPLPS cost functions using even just relaxed consistencies. The results are useful when we have cost functions whose minimum cost computation is polynomial time but that for conjunctions of such cost functions is not. We show that *flow-based projection safe* [28, 30] and *polynomially decomposable* [31] cost functions are IPLPS, in which the minimum cost computation is NP-hard for the conjunctions of an important subclass of them.

Fifth, we demonstrate the practicality of our framework with empirical results. We conduct experiments on several examples of polynomially linear projection-safe cost functions and integral polynomially linear projection-safe cost functions, together with their conjunctions, against the flow-based and polynomially decomposable approaches as well as pure integer programming. We observe orders of magnitude in runtime and search space improvements when the conjunctions of PLPS or IPLPS cost functions are used together with relaxed consistencies and the results agree with our theorems.

6.2 Future Work

We have introduced the concept of polynomially linear projection-safe cost functions and integral polynomially linear projection-safe cost functions, together with relaxed consistencies. They give at least three possibilities for future work.

The first possible question is whether we can enhance the relaxed consistencies for stronger consistency notions like optimal soft arc consistency (OSAC) [14, 13], virtual arc consistency (VAC) [12, 13] and k-consistency [11]. Currently, we only give the relaxed versions of star generalized arc consistency (GAC*) [43], full star generalized directional arc consistency (FDAC*) [28, 30] and weak star existential directional generalized arc consistency (weak EDGAC*) [29, 30]. They can be reformulated such that their major conditions are represented with the minimum costs of cost functions. Those consistency notions can be relaxed by replacing the

minimum costs into approximated minimum costs in their conditions. It might not be straight forward to relax the consistency notions with different kinds of conditions and involving rational costs like OSAC and VAC. It is interesting to see if there are different ways to relax the consistency notions which can also tackle those consistency notions.

The second possible question is whether we can give necessary conditions for polynomially linear projection-safety and other $\mathcal{T}$ projection-safety. Currently, we only give the sufficient conditions for polynomially linear projection-safety. The necessary conditions of polynomially linear projection-safety may allow us to find out whether the other kinds of useful global cost functions are PLPS or not.

We also observe that the effects of enforcing relaxed consistencies on some polynomially linear projection-safe cost functions can be very different from that of enforcing standard consistencies on them. For example, enforcing relaxed consistencies on the SOFT_NVALUE cost functions may have little or no effect since the minimum cost arising from the minimum number of values needed cannot be approximated. The third possible question is whether we can give conditions to identify such kind of cost functions and suggest practical ways to handle them.

Bibliography

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling and Placement Problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[2] N. Beldiceanu, M. Carlsson, and J. Rampon. Global Constraint Catalog. SICS Research Report, 2005.

[3] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[4] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and D. Vincent. Experiments in Mixed Integer Linear Programming. *Mathematical Programming*, 1(1):76–94, 1971.

[5] T. Berthold. Primal Heuristics for Mixed Integer Programs. Master's thesis, Technische Universität Berlin, 2006.

[6] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The SLIDE Meta-Constraint. Technical report, 2007.

[7] C. Bessière and P. V. Hentenryck. To Be or Not to Be . . . a Global Constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, pages 789–794, 2003.

[8] C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh. Propagating Conjunctions of ALLDIFFERENT Constraints. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 27–32, 2010.

[9] C. Bessière and J.-C. Régin. Arc Consistency for General Constraint Networks: Preliminary Results. In *Proceedings of the 15th International Joint Conferences on Artificial Intelligence*, pages 398–404, 1997.

[10] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. *Constraints*, 4(1):79–89, 1999.

[11] M. C. Cooper. High-Order Consistency in Valued Constraint Satisfaction. *Constraints*, 10(3):283–305, 2005.

[12] M. C. Cooper, S. de Givry, M. Sànchez, T. Schiex, and M. Zytnicki. Virtual Arc Consistency for Weighted CSP. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 253–258, 2008.

[13] M. C. Cooper, S. de Givry, M. Sànchez, T. Schiex, M. Zytnicki, and T. Werner. Soft Arc Consistency Revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.

[14] M. C. Cooper, S. de Givry, and T. Schiex. Optimal Soft Arc Consistency. In *Proceedings of the 20th International Joint Conferences on Artificial Intelligence*, pages 68–73, 2007.

[15] M. C. Cooper and T. Schiex. Arc Consistency for Soft Constraints. *Artificial Intelligence*, 154(1-2):199–227, 2004.

[16] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[17] S. de Givry, F. Heras, M. Zytnicki, and J. Larrosa. Existential Arc Consistency: Getting Closer to Full Arc Consistency in Weighted CSPs. In *Proceedings of*

the 19th International Joint Conferences on Artificial Intelligence, pages 84–89, 2005.

[18] H. Fargier and J. Lang. Uncertainty in Constraint Satisfaction Problems: a Probabilistic Approach. In *Proceedings of the 2nd European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 97–104, 1993.

[19] E. C. Freuder and R. J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.

[20] J. N. Hooker. *Integrated Methods for Optimization*. Springer Science + Business Media, LLC, 2007.

[21] IBM. IBM ILOG CPLEX Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[22] I. Katriel and S. Thiel. Complete Bound Consistency for the Global Cardinality Constraint. *Constraints*, 10(3):115–135, 2005.

[23] A. M. Koster. *Frequency Assignment: Models and Algorithms*. PhD thesis, University of Maastricht, 1999.

[24] J. Larrosa. Node and Arc Consistency in Weighted CSP. In *Proceedings of the 18th AAAI Conference on Artificial Intelligence*, pages 48–53, 2002.

[25] J. Larrosa. In the Quest of the Best Form of Local Consistency for Weighted CSP. In *Proceedings of the 18th International Joint Conferences on Artificial Intelligence*, pages 239–244, 2003.

[26] J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc Consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.

[27] J.-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[28] J. H. M. Lee and K. L. Leung. Towards Efficient Consistency Enforcement for Global Constraints in Weighted Constraint Satisfaction. In *Proceedings of the 21st International Joint Conferences on Artificial Intelligence*, pages 559–565, 2009.

[29] J. H. M. Lee and K. L. Leung. A Stronger Consistency for Soft Global Constraints in Weighted Constraint Satisfaction. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 121–127, 2010.

[30] J. H. M. Lee and K. L. Leung. Consistency Techniques for Flow-Based Projection-Safe Global Cost Functions in Weighted Constraint Satisfaction. *Journal of Artificial Intelligence Research*, 43:257–292, 2012.

[31] J. H. M. Lee, K. L. Leung, and Y. Wu. Polynomially Decomposable Global Cost Functions in Weighted Constraint Satisfaction. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, pages 507–513, 2012.

[32] J. H. M. Lee and Y. W. Shum. Modeling Soft Global Constraints as Linear Programs in Weighted Constraint Satisfaction. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence*, pages 305–312, 2011.

[33] A. K. Mackworkth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[34] M.J. Maher, N. Narodytska, C.-G. Quimper, and T. Walsh. Flow-Based Propagators for the SEQUENCE and Related Global Constraints. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, pages 159–174, 2008.

[35] H. Marchand and L. A. Wolsey. Aggregation and Mixed Integer Rounding to Solve MIPs. *Operations Research*, 49(3):363–371, 2001.

[36] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[37] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.

[38] G. Pesant, C.-G. Quimper, L.-M. Rousseau, and M. Sellmann. The Polytope of Context-Free Grammar Constraints. In *Proceedings of the 8th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*, pages 29–43, 2009.

[39] T. Petit, J.-C. Régin, and C. Bessière. Meta-Constraints on Violations for Over Constrained Problems. In *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence*, pages 358–365, 2000.

[40] T. Petit, J.-C. Régin, and C. Bessière. Specific Filtering Algorithm for Over-Constrained Problems. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 451–463, 2001.

[41] J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraints. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, pages 209–215, 1996.

[42] J.-C. Régin. Combination of Among and Cardinality Constraints. In *Proceedings of the 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 288–303, 2005.

[43] M. Sànchez, S. de Givry, and T. Schiex. Mendelian Error Detection in Complex Pedigrees Using Weighted Constraint Satisfaction Techniques. *Constraints*, 13(1-2):130–154, 2008.

[44] T. Sandholm. An Algorithm for Optimal Winner Determination in Combinatorial Auctions. In *Proceedings of the 16th International Joint Conferences on Artificial Intelligence*, pages 542–547, 1999.

[45] T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In Chris Mellish, editor, *Proceedings of the 14th International Joint Conferences on Artificial Intelligence*, pages 631–639, 1995.

[46] L. Shapiro and R. Haralick. Structural Descriptions and Inexact Matching. *IEEE Transactions Pattern Analysis Machine Intelligence*, 3(5):504–519, 1981.

[47] C. Solnon, V. Cung, A. Nguyen, and C. Artigues. The Car Sequencing Problem: Overview of State-of-the-Art Methods and Industrial Case-Study of the ROADDEF'2005 Challenge Problem. *European Journal of Operational Research*, 191(3):912–927, 2008.

[48] W. van Hoeve, G. Pesant, and L. Rousseau. On Global Warming: Flow-Based Soft Global Constraints. *Journal of Heuristics*, 12(4-5):347–373, 2006.

[49] L. Wolsey. *Integer Programming*. Wiley, 1998.

[50] M. Zytnicki, C. Gaspin, and T. Schiex. A New Local Consistency for Weighted CSP Dedicated to Long Domains. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 394–398, 2006.