Theoretical Properties and Efficient Satisfaction of Hierarchical Constraint Systems
階層制約系の理論的性質と効率的解消法

Version 1.0.2

by
Hiroshi Hosobe
細部 博史

A Dissertation

Submitted to
The Graduate School of
The University of Tokyo
in Partial Fulfillment of the Requirements
for The Degree of Doctor of Science
in Information Science

December 1997

Abstract

Constraints are recognized as powerful tools for various problems such as management of knowledge, logic programming, and construction of graphical user interfaces. Especially, hierarchical constraint systems (HCSs) are promising since they are effective in modeling over-constrained real-world problems. However, there are a small number of practical systems and applications adopting HCSs, because few efficient and reliable constraint solvers for HCSs are available. A major reason for this situation is that properties of HCSs useful in designing efficient constraint satisfaction algorithms are unclear.

The aim of this research is to (1) explore theoretical properties of HCSs and provide foundations of how to design algorithms for satisfying HCSs. Furthermore, we (2) prove the viability of our new foundations by actually developing two constraint solvers for HCSs, mainly targeting construction of graphical user interfaces.

In this research, we treat two kinds of HCSs. First, we focus on the theory of constraint hierarchies, which is one of the most popular formulations of HCSs. To begin with, we reformulate the theory with a more strict definition. Next, we propose generalized local propagation (GLP) as a framework for studying constraint hierarchies, and show properties useful for constraint satisfaction. Then, applying the result of GLP, we develop the *DETAIL* constraint solver, whose algorithm is the first local propagation method that solves constraint hierarchies with a global criterion.

As another novel formulation of HCSs, we propose hierarchical linear systems (HLSs), which can be viewed as specialization of constraint hierarchies in linear constraints. HLSs provide a basis for designing numerical constraint satisfaction algorithms that are more reliable than existing local propagation algorithms. Finally, using HLSs, we develop the HiRise constraint solver, integrating the advantages of local propagation with its numerical algorithm.

要旨

制約は知識処理，論理型プログラミング，グラフィカルユーザーインターフェースの構築など，様々な問題のための強力な道具として認識されている．特に階層制約系は，制約過多である実世界の問題のモデリングに適しているため，有望である．しかし，階層制約系のための効率的で信頼性の高い制約解消系が少ないため，実用的なシステムやアプリケーションの数は少ない．その主な原因は，効率的解消法の設計に役立つような階層制約系の性質が明らかでないという点にある．

本研究の目的は，(1) 階層制約系の理論的性質を探求し，階層制約系の解消法を設計するための基礎を与えることである．さらに，(2) 主に GUI の構築を対象として，階層制約系のための 2 つの制約解消系を実際に開発することにより，その新しい基礎の能力を立証する．

本研究では 2 種類の階層制約系を扱う．最初に，最も広く研究されている階層制約系の 1 つである制約階層の理論に焦点を当てる．まず，その理論をより厳密に再定式化する．次に，制約階層を研究するための枠組として一般化局所伝播法を提案し，制約解消に有用な性質を示す．そして，一般化局所伝播法の結果を応用し，*DETAIL*制約解消系を開発する．そのアルゴリズムは，大域的な基準で制約階層を解消する最初の局所伝播法である．

階層制約系の別の新しい定式化として，階層線形系を提案する．階層線形系は，制約階層を線形制約に特化したものと見なすことができ，既存の局所伝播法アルゴリズムより信頼性の高い数値的制約解消アルゴリズムを設計するための基礎を与える．最後に，階層線形系を利用して，HiRise 制約解消系を開発し，局所伝播法の利点を数値アルゴリズムに統合する．

Acknowledgment

I wish to express my deep gratitude to my supervisor Prof. Akinori Yonezawa, who invariably supported and encouraged me during this research. I am greatly indebted to Prof. Satoshi Matsuoka for continuously guiding and helping me. I am grateful to Mr. Shin Takahashi and Mr. Ken Miyashita for their collaboration on my early work. Finally, I thank all the members of TRIP Group (a joint user interface group at the University of Tokyo and the Tokyo Institute of Technology) for plenty of comments and suggestions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraints are recognized as powerful tools for various problems such as management of knowledge, logic programming, and construction of graphical user interfaces (GUIs). With constraints, programmers and even end users can solve problems easily since they have only to describe what are problems, instead of dictating how to solve problems with traditional procedures.

In this dissertation, we discuss constraints from both of the theoretical and practical viewpoints. In practical discussion, we mainly focus on construction of GUIs.

1.1 Background

This section presents the background of constraints especially from the viewpoint of GUIs.

1.1.1 Using Constraints for Construction of Graphical User Interfaces

Generally, *constraints* represent relations among variables that should be maintained. When declared, a set of constraints works together as a *constraint system*,[1] and expresses the total relationship among all the variables that they constrain. To realize this process, software components called *constraint solvers* automatically compute *solutions* of constraint systems by providing variables in the systems with appropriate values. For an ordinary

[1] Constraint systems are also known as *constraint satisfaction problems* and *constraint networks*.

constraint system, each of its solutions is determined so that it satisfies all the constraints in the system.

In construction of GUIs, programmers employ constraints to represent relations among internal data, connections between internal data and graphical objects, and layouts of graphical objects. Consider, for example, that a programmer tries to layout four objects a, b, c, and d by specifying the following constraint system:

$$\begin{aligned} \mathsf{a.y} &= \mathsf{b.y} \\ \mathsf{c.y} &= \mathsf{d.y} \\ \mathsf{a.x} &= \mathsf{c.x} \\ \mathsf{b.x} &= \mathsf{d.x}. \end{aligned} \tag{1.1}$$

Then, satisfying the constraints, the constraint solver may obtain a solution indicating the rectangular structure of the objects, for example, as illustrated in Figure 1.1.

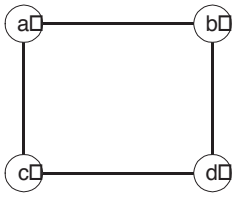

Figure 1.1: Layouting four objects in a rectangle.

For GUIs, constraints are useful for specifying dynamic behavior as well as static structure such as the rectangular layout of the objects. For instance, suppose that the programmer wants to move the overall rectangle in the previous example by dragging the object d with a mouse. With constraints, the programmer can accomplish this task by adding the following new constraints, which bind the mouse cursor with d:

$$\begin{aligned} \mathsf{d.x} &= \mathsf{mouse.x} \\ \mathsf{d.y} &= \mathsf{mouse.y}. \end{aligned} \tag{1.2}$$

While d is being dragged, the automatic nature of the constraint solver will maintain the constraint system, which may result in the motion of the overall rectangular structure as shown in Figure 1.2 (where the gray image indicates the layout before the drag). To release d, the programmer only needs to remove the mouse binding constraint from the system. A point is that the

programmer did not have to consider which objects to move later when he or she specified the constraint system (1.1) for the layout. Whichever object the programmer binds the mouse cursor with, the constraint solver will maintain the resulting constraint system to obtain an appropriate behavior.

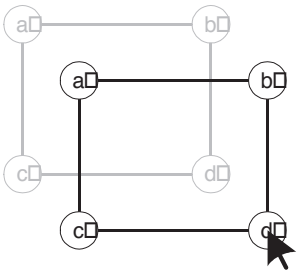

Figure 1.2: Dragging an object in a rectangular layout.

1.1.2 Problems of Under- and Over-Constrained Systems

Constraint systems may have multiple solutions, and such systems are said to be *under-constrained*. Programmers often suffer from under-constrained systems because some of their solutions do not satisfy their intention. As an example, consider again the previous example of layouting objects in a rectangle. For the constraint system (1.1) with (1.2), the constraint solver may obtain other layouts as given in Figures 1.3 (a) and 1.3 (b). In Figure 1.3 (a), the object a stays where it was, and thus the rectangle is resized as d is moved. By contrast, in Figure 1.3 (b), although neither of the size of the rectangle and the positions of the objects are preserved, the behavior is correct as a solution of the constraint system (but is almost unlikely to be acceptable to the programmer).

To avoid undesirable solutions due to under-constrained situations, programmers must present sufficient sets of constraints to systems. In the layout example, to move the overall rectangle without changing its size, the programmer should have provided in advance constraints fixing the size, e.g.

$$\begin{array}{l} \mathit{stay}(\mathsf{width}) \\ \mathit{stay}(\mathsf{height}) \end{array} \tag{1.3}$$

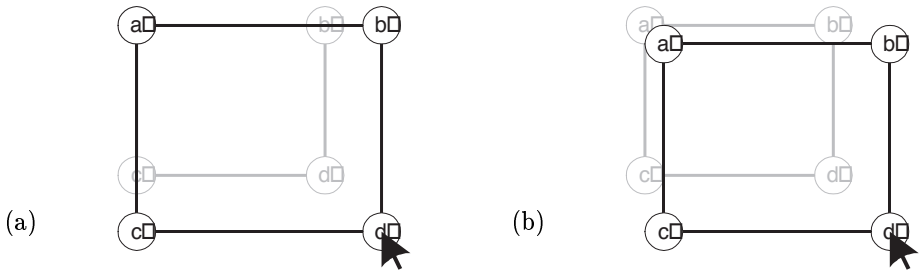

Figure 1.3: Possible layouts resulting from the under-constrained system.

where we assume that the following constraints are also specified:

$$\begin{array}{c} \mathsf{a.x} + \mathsf{width} = \mathsf{b.x} \\ \mathsf{a.y} + \mathsf{height} = \mathsf{c.y}. \end{array} \tag{1.4}$$

Ordinary constraint systems sometimes have no solutions because they have too many constraints, and are said to be *over-constrained*. Again in the layout example, consider that the programmer changes his or her plan to resizing the rectangle although he or she previously supplied the constraints (1.3) for moving the overall layout. Then the programmer may declare constraints that stay the object a as follows:

$$\begin{array}{c} \mathit{stay}(\mathsf{a.x}) \\ \mathit{stay}(\mathsf{a.y}). \end{array} \tag{1.5}$$

However, when the object d is dragged with (1.2), the constraint solver will fail because of the conflict of (1.5) with the previous constraints (1.3) for fixing the size.

1.1.3 Constraint Hierarchies

To prevent constraint systems from getting under-constrained or over-constrained, it is required for programmers to maintain necessary and sufficient sets of constraints. However, it is difficult to always guarantee this requirement. Therefore, we need some mechanism for appropriately treating over-constrained situations.

For this purpose, various approaches are proposed. In particular, '*constraint hierarchy*,' which is a class of hierarchical constraint systems (HCSs), is promising for GUIs. By definition, a constraint hierarchy consists of constraints with hierarchical *strengths* such as required, strong, medium, and

weak, which can be regarded as the preferences or priorities of the constraints. Intuitively, solutions of constraint hierarchies are determined so that they will satisfy as many strong constraints as possible, leaving weaker inconsistent ones unsatisfied. In other words, weak constraints work by default when stronger conflicting ones do not exist.

In constructing GUIs with constraint hierarchies, programmers use weaker constraints for specifying relations that they may revoke. For instance, in the running layout example, the programmer initially gives (1.1) (for the rectangular structure) and (1.4) (for the size) as required constraints and (1.3) (for fixing the size) as weak. Then, when he or she supplies (1.2) (for dragging d) as medium, the rectangle will be moved as is with an ordinary constraint system. By contrast, if the programmer provides (1.5) (for staying a) as strong in addition to (1.2) as medium, the rectangle will be resized, revoking the weak constraints (1.3), as opposed to the case with an ordinary system that resulted in a failure.

1.1.4 Satisfaction of Constraint Hierarchies

To realize interactive GUIs using constraint hierarchies, we need efficient constraint satisfaction algorithms that handle constraints expressive enough to construct GUIs. Constraint hierarchies are unique among approaches to over-constrained systems in that they have efficient constraint satisfaction algorithms proposed. We can categorize the algorithms into the following three approaches:

The refining method first satisfies the strongest level, and then, weaker levels successively. It is mainly employed in constraint logic programming languages such as HLCP(R, $\star$) with the DeltaStar constraint solver [90] and CHAL [77].

The optimization approach transforms constraint hierarchies into optimization problems by assigning appropriate weights to strengths. It is adopted in recent constraint solvers for GUIs such as Cassowary and QOCA [12].

Local propagation gradually solves hierarchies by repeatedly selecting uniquely satisfiable constraints. It is used in various constraint solvers for GUIs including DeltaBlue [22] and SkyBlue [73].

First, to see the refining method, suppose we have the following constraint hierarchy:

$$\begin{array}{ll} \mathsf{required} & x = y \\ \mathsf{strong} & y = z + 1 \\ \mathsf{medium} & z = 0 \\ \mathsf{weak} & z = 1. \end{array}$$

This is solved as follows: First, by satisfying the strongest constraint required $x = y$, the method reduces the set $\boldsymbol{\Theta}$ of all variable assignments (mappings from variables to their values) to

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(x) = \theta(y)\}.$$

Second, by fulfilling the next strongest one strong $y = z + 1$, it obtains

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(x) = \theta(y) \wedge \theta(y) = \theta(z) + 1\}.$$

Third, evaluating medium $z = 0$ yields

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(x) = 1 \wedge \theta(y) = 1 \wedge \theta(z) = 0\}.$$

Now, the weakest constraint weak $z = 1$ conflicts with the assignments that have been generated from the stronger constraints, and therefore, remains unsatisfied. As shown in this example, the refining method is a 'straightforward' algorithm for solving constraint hierarchies.

Next, to make out the optimization approach, reconsider the hierarchy in the last example. It transforms the hierarchy into the following optimization problem:

$$\begin{array}{rl} \text{minimize} & w_{\mathsf{strong}} e(\varepsilon_1) + w_{\mathsf{medium}} e(\varepsilon_2) + w_{\mathsf{weak}} e(\varepsilon_3) \\ \text{subject to} & x = y \\ & y = z + 1 + \varepsilon_1 \\ & z = 0 + \varepsilon_2 \\ & z = 1 + \varepsilon_3 \end{array}$$

where w_{strong}, w_{medium}, and w_{weak} indicate weights associated with the strengths strong, medium, and weak respectively. Weights are determined so that one for a strength is heavier than ones for weaker strengths. Also, the function e is usually defined as $e(\varepsilon) = |\varepsilon|$ or $e(\varepsilon) = \varepsilon^2$. Intuitively, $e(\varepsilon_1)$, $e(\varepsilon_2)$, and $e(\varepsilon_3)$ represent errors of $y = z + 1$, $z = 0$, and $z = 1$ respectively. Algorithms adopting this approach solve such optimization problems

using some known optimization techniques. It depends on actual algorithms whether they can solve constraint hierarchies exactly. In other words, some algorithms obtain approximate solutions.

Finally, to comprehend local propagation, consider the last constraint hierarchy again. Local propagation handles it as follows: First, since medium $z = 0$ can be uniquely solved, it acquires the set of variable assignments

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(z) = 0\}.$$

Next, since the instantiation of z makes strong $y = z + 1$ uniquely satisfiable, it produces

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(y) = 1 \wedge \theta(z) = 0\}.$$

Finally, computing required $x = y$, it outputs

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(x) = 1 \wedge \theta(y) = 1 \wedge \theta(z) = 0\}.$$

Note it must reject the weakest constraint weak $z = 1$ at the beginning; otherwise, it would yield an incorrect or empty solution. As suggested with this example, local propagation algorithms must *schedule* constraints, or plan in what order they will choose and solve constraints, discarding the ones that lead to incorrect solutions.

1.2 Our Goal, Approach, and Contributions

This section describes our approach to the goal of this research, and our contributions to the research area.

1.2.1 Our Goal and Approach

The goal of this research is to reveal theoretical properties of HCSs and provide foundations of how to design algorithms for satisfying HCSs. Furthermore, we prove the viability of our new foundations by actually developing two constraint solvers for HCSs, mainly targeting construction of GUIs.

In investigating properties of HCSs, we direct our attention to local propagation, which is largely different from the other approaches. While it is rather easy to intuitively understand how the refining method and the optimization approach work correctly, it is difficult to understand how local propagation guarantees that it obtains correct solutions of HCSs.

As a first step, we focus on the theory of constraint hierarchies, which is, as noted, one of the most popular formulations of HCSs. To begin with, we

reformulate the theory with a more strict definition [38]. Next, we propose generalized local propagation (GLP) as a framework for studying constraint hierarchies, and show properties useful for constraint satisfaction [36, 38]. Then applying the result of GLP, we develop the *DETAIL* constraint solver, whose algorithm is the first local propagation method that solves constraint hierarchies with a global criterion [39, 40].

Past local propagation algorithms treat dataflow constraints, although most practical applications demand algebraic constraints. For ordinary (i.e. non-hierarchical) constraint systems, algebraic constraints are usually solved with numerical algorithms. Also, some HCSs can be addressed with the refining method or the optimization approach.

In the second stage of this research, we discuss how to solve HCSs with algebraic constraints by introducing the essential idea of local propagation into numerical algorithms. This does not only expand the range of types of constraints that can be handled with local propagation, but also allows us to adopt various useful techniques developed in the field of numerical computation to solve HCSs.

Specifically, we propose a novel formulation of HCSs that we call hierarchical linear systems (HLSs), which can be viewed as a specialization of constraint hierarchies in linear constraints [37]. HLSs provide a basis for designing numerical constraint satisfaction algorithms that are more reliable than existing local propagation algorithms. Finally, using HLSs, we develop the HiRise constraint solver, integrating the advantage of local propagation with its numerical algorithm.

1.2.2 Our Contributions

Contributions of this research can be summarized as follows:

- We present an alternative formulation of constraint hierarchies that is more strict and abstract than previous operational ones. With this formulation, we can discuss properties of criteria for satisfying constraint hierarchies from a more general viewpoint. In fact, we construct an important class that contains most of known useful criteria.

- We provide GLP as a theoretical framework for investigating satisfaction of constraint hierarchies. It covers approaches that solve constraint hierarchies by dividing them. For example, it can explain both the refining method and local propagation.

- We develop the *DETAIL* constraint solver based on GLP. It is the first

local propagation solver that maintains constraint hierarchies with a global criterion.

- We propose HLSs as a simple but useful class of HCSs. With this, we show that HCSs are not special but natural concepts that are easy to understand. We also suggest that various existing methods for linear computation will be easily applicable to satisfaction of HLSs.
- We develop the HiRise constraint solver by adopting HLSs. It realizes incremental constraint satisfaction by incorporating local propagation technology into a numerical algorithm. It is a significant contribution since it proves that local propagation is useful even for numerical algorithms.

1.3 Overview of the Dissertation

This dissertation is composed as follows:

- Chapter 2 formalizes constraint hierarchies. First, it introduces the original theory of constraint hierarchies. Next, it proposes our modified theory, and finally discusses the difference between the original and ours.
- Chapter 3 proposes generalized local propagation (GLP) as a theoretical framework for scheduling constraints in constraint hierarchies.
- Chapter 4 presents a constraint solver called *DETAIL*, which is the first local propagation algorithm adopting a global criterion.
- Chapter 5 formalizes hierarchical linear systems (HLSs) as a specialization of constraint hierarchies in linear constraints, and provides several basic algorithms for solving HLSs.
- Chapter 6 describes the HiRise constraint solver, which provides incremental planning and real-time execution for satisfying HLSs.
- Chapter 7 describes previous researches on constraints, especially from the viewpoints of treatment and satisfaction of over-constrained systems.
- Chapter 8 concludes the dissertation.

Chapter 2

Constraint Hierarchies

This chapter formalizes constraint hierarchies. First, it introduces the original theory of constraint hierarchies. Next, it proposes our modified theory, and finally discusses the difference between the original and ours.

2.1 The Original Theory of Constraint Hierarchies

This section gives the original theory of constraint hierarchies proposed by Borning et al. They first presented a formulation of constraint hierarchies in [7], and later provided revised formulations [9, 11, 90, 91, 92, 93]. This section introduces their formulation based on [90].[1]

2.1.1 Formulation

Let $\boldsymbol{X}$ be the set of variables, $\boldsymbol{D}$ the domain of the variables, and $\boldsymbol{C}$ the set of constraints. A strength is an integer between 0 and l_{w}, where l_{w} is some positive integer. Intuitively, strength 0 indicates the required strength, strengths larger than 0 represent preferential strengths, and the larger the number of a strength, the weaker it is. A *labeled constraint* is a constraint associated with a strength.

A constraint hierarchy is a finite multi-set of labeled constraints. Given a constraint hierarchy H, H is divided into levels $H_0, H_1, \ldots, H_{l_{\mathrm{w}}}$, where H_l is a sequence of constraints with strength l in some arbitrary order, e.g.

$$H_l = [c_1, c_2, \ldots, c_k]$$

[1] Some technical terms and mathematical symbols are modified for consistency with our theory in the next section.

where each c_i is a constraint in level l of H.

Solutions of a constraint hierarchy are expressed as *variable assignments*.[2] A variable assignment θ is a mapping from $\boldsymbol{X}$ to $\boldsymbol{D}$, and $\boldsymbol{\Theta}$ indicates the set of all variable assignments.

To define solutions, the original formulation uses a comparator denoted as *better*. Intuitively, $better(\theta, \theta', H)$ means that θ is better than θ' according to H.

Now, the solution set S of a constraint hierarchy H is defined as follows:

$$S = \{\theta \in S_0 \mid \neg\exists\theta' \in S_0.\, better(\theta', \theta, H)\}$$

where

$$S_0 = \{\theta \in \boldsymbol{\Theta} \mid \forall c \in H_0.\, holds(c, \theta)\}$$

in which $holds(c, \theta)$ means that c is exactly satisfied for θ. Intuitively, S_0 is the set of all variable assignments satisfying the required constraints in H, and a solution of H is an assignment in S_0 that has no better assignments in S_0.

The original formulation insists that all *better* comparators are irreflexive:

$$\forall H.\, \forall\theta.\, \neg\, better(\theta, \theta, H).$$

Also, it assumes that most (not all) *better* comparators are transitive:

$$\forall H.\, \forall\theta.\, \forall\theta'.\, \forall\theta''.\, better(\theta, \theta', H) \wedge better(\theta', \theta'', H) \Rightarrow better(\theta, \theta'', H).$$

It insists another property "*better* respects the hierarchy," that is, if there is a variable assignment in S_0 that completely satisfies all the constraints through level l, then all variable assignments in S must satisfy all the constraints through level l:

$$\begin{aligned}(\exists\theta \in S_0.\, \exists l > 0.\, \forall l' \in \{1, \ldots, l\}.\, \forall c \in H_l.\, holds(c, \theta))\\ \Rightarrow\quad \forall\theta' \in S.\, \forall i \in \{1, \ldots, l\}.\, \forall c \in H_l.\, holds(c, \theta').\end{aligned}$$

[2] The original theory employs the term 'valuation' instead of 'variable assignment.' However, in this dissertation, we use 'variable assignment' since it seems to be more familiar.

2.1.2 Definition of Comparators

This subsection defines *better* comparators, and presents a brief example of satisfaction of a constraint hierarchy.

Given a constraint c and a variable assignment θ, an *error function* $e(c, \theta)$ returns a non-negative real number by evaluating the error of c for θ. The condition $e(c, \theta) = 0$ indicates that c is exactly satisfied for θ. The original formulation provides two kinds of error functions: *predicate* and *metric*. The predicate error function is defined as follows:

$$e(c, \theta) = \begin{cases} 0 & \text{if } c \text{ is exactly satisfied for } \theta \\ 1 & \text{otherwise.} \end{cases}$$

By contrast, the metric error function is calculated using some distance. For example, the error of a constraint $x = y$ is defined as

$$e(\text{`}x = y\text{'}, \theta) = |\theta(x) - \theta(y)|.$$

Given a level H_l and a variable assignment θ, the function E returns the sequence containing errors of constraints in H_l for θ:

$$E(H_l, \theta) = [e(c_1, \theta), e(c_2, \theta), \ldots, e(c_k, \theta)].$$

A *combining error function* g joins errors $E(H_l, \theta)$ into a *combined error*. Two combined errors are compared by a reflexive and symmetric relation $<>_g$ and an irreflexive, antisymmetric, and transitive relation $<_g$. Intuitively, $g(E(H_l, \theta)) <>_g g(E(H_l, \theta'))$ means that the combined error of H_l for θ is similar to the one for θ', and $g(E(H_l, \theta)) <_g g(E(H_l, \theta'))$ indicates that the combined error for θ is smaller than the one for θ'. The original formulation proposes several comparators by presenting instances of $<>_g$ and $<_g$, which will be described in the next subsection.

A *combined error sequence* of H is a sequence with combined errors of all preferential levels, i.e.

$$[g(E(H_1, \theta)), g(E(H_2, \theta)), \ldots, g(E(H_{l_w}, \theta))].$$

Two combined error sequences $[\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_{l_w}]$ and $[\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{l_w}]$ are compared by a lexicographic order $<_G$ as follows:

$$\begin{aligned} &[\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_{l_w}] <_G [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{l_w}] \\ &\quad \equiv \quad \exists l \in \{1, \ldots, l_w\}. (\forall l' \in \{1, \ldots, l-1\}. \boldsymbol{u}_{l'} <>_g \boldsymbol{v}_{l'}) \wedge \boldsymbol{u}_l <_g \boldsymbol{v}_l. \end{aligned}$$

Now, *better* is defined as follows:

$$\begin{aligned} &better(\theta, \theta', H) \\ &\quad \equiv \quad [g(E(H_1, \theta)), \ldots, g(E(H_{l_w}, \theta))] <_G [g(E(H_1, \theta')), \ldots, g(E(H_{l_w}, \theta'))]. \end{aligned}$$

Since the comparator is defined as lexicographic ordering with combined errors of levels as its components, the result of a level has absolute priority over those of weaker ones.

To see how the formulation works, consider the following simple example of a constraint hierarchy quoted from [90]:

required	$x > 0$
strong	$x < 10$
weak	$x = 4$

where the domain is assumed to be real. First, the set S_0 is determined as the set of all variable assignments satisfying the required constraint $x > 0$, i.e.

$$S_0 = \{\theta \in \boldsymbol{\Theta} \mid \theta(x) > 0\}.$$

Then the solution set S is made as the set of all assignments in S_0 that have no better assignments in S_0. Since *better* is lexicographic ordering, it respects the strong constraint $x < 10$ more than the weak constraint $x = 4$. Therefore, S_0 can be narrowed into

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(x) > 0 \wedge \theta(x) < 10\}$$

which consists of assignments in S_0 that satisfy $x < 10$. Finally, using the weak constraint $x = 4$, S is obtained as follows:

$$\begin{aligned} S &= \{\theta \in \boldsymbol{\Theta} \mid \theta(x) > 0 \wedge \theta(x) < 10 \wedge \theta(x) = 4\} \\ &= \{\theta \in \boldsymbol{\Theta} \mid \theta(x) = 4\}. \end{aligned}$$

2.1.3 Instances of Comparators

Most constraint hierarchies have multiple constraints at each levels. It follows that constraints may conflict with each other inside a single level. To treat conflicts inside levels, constraint hierarchies use various comparators defined with $<>_g$ and $<_g$.

The original formulation classifies comparators into three categories: *globally-better*, *locally-better*, and *regionally-better*. Let $\boldsymbol{u} = [u_1, u_2, \ldots, u_k]$

and $\boldsymbol{v} = [v_1, v_2, \ldots, v_k]$ be sequences of errors of constraints. For globally-better, $<>_g$ and $<_g$ are defined with the following forms:

$$\begin{aligned} \boldsymbol{u} <>_g \boldsymbol{v} &\equiv g(\boldsymbol{u}) = g(\boldsymbol{v}) \\ \boldsymbol{u} <_g \boldsymbol{v} &\equiv g(\boldsymbol{u}) < g(\boldsymbol{v}) \end{aligned}$$

where the combining error function g returns a non-negative real number, and $=$ and $<$ are the ones for real numbers. Several instances of globally-better are proposed, and major ones are *weighted-sum-better*, *least-squares-better*, and *worst-case-better*:

- For weighted-sum-better, the combining function is

  $$g(\boldsymbol{v}) \equiv \sum_i w_i v_i$$

  where w_i indicates the weight associated with the i-th constraint.

- For least-squares-better, the combining function is

  $$g(\boldsymbol{v}) \equiv \sum_i w_i v_i^2.$$

- For worst-case-better, the combining function is

  $$g(\boldsymbol{v}) \equiv \max_i \{w_i v_i\}.$$

For each comparator, error functions may be either predicate or metric. For example, *weighted-sum-predicate-better* is defined as weighted-sum-better using the predicate error function, and *weighted-sum-metric-better* is defined as weighted-sum-better with the metric one. Also, as a special case of weighted-sum-predicate-better, *unsatisfied-count-better* is defined using weight 1 for each constraint; intuitively, unsatisfied-count-better compares assignments using the number of unsatisfied constraints.

For locally-better, $<>_g$ and $<_g$ are defined as follows:

$$\boldsymbol{u} <>_g \boldsymbol{v} \equiv \forall i. u_i = v_i \tag{2.1}$$

$$\boldsymbol{u} <_g \boldsymbol{v} \equiv \forall i. u_i \leq v_i \wedge \exists i'. u_{i'} < v_{i'}. \tag{2.2}$$

Locally-better using the predicate error function is called *locally-predicate-better*, and the one with the metric function is called *locally-metric-better*.[3]

[3] Locally-metric-better is also known as *locally-error-better* [5].

Locally-better considers each constraint individually. It is often unable to compare variable assignments because of the situation that one assignment produces an error smaller than the other for some constraint but larger for another constraint.

For regionally-better, $<_g$ and $<>_g$ are defined as follows:

$$\begin{aligned} \boldsymbol{u} <_g \boldsymbol{v} &\equiv \forall i. u_i \leq v_i \wedge \exists i'. u_{i'} < v_{i'} \\ \boldsymbol{u} <>_g \boldsymbol{v} &\equiv \neg(\boldsymbol{u} <_g \boldsymbol{v} \vee \boldsymbol{v} <_g \boldsymbol{u}). \end{aligned}$$

Regionally-better has the same $<_g$ as locally-better, but it is always able to compare variable assignments because of the different definition of $<>_g$. As a result, regionally-better tends to yield solution sets smaller than locally-better. Also, it should be noted that regionally-better is not transitive while globally-better and locally-better are transitive.

To make out how comparators work differently, consider the following constraint hierarchy:

$$\begin{array}{ll} \mathsf{required} & x > 0 \\ \mathsf{strong} & x = 2 \\ \mathsf{strong} & x = 4 \end{array}$$

where both of the strong constraints have weight 1 if needed. Then, with weighted-sum-metric-better, the solution set is

$$\{\theta \in \boldsymbol{\Theta} \mid 2 \leq \theta(x) \leq 4\}.$$

With least-squares-better and worst-case-better, the solution set is

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(x) = 3\}.$$

With weighted-sum-predicate-better, locally-better, and regionally-better, the solution set is

$$\{\theta \in \boldsymbol{\Theta} \mid \theta(x) = 2 \vee \theta(x) = 4\}.$$

To further learn the difference among comparators, see [9] and [90].

2.2 Our Theory of Constraint Hierarchies

This section provides our theory of constraint hierarchies.

2.2.1 Formulation

First, we modify the original formulation of constraint hierarchies so that it will allow us to better investigate properties of constraint hierarchies. Intuitively, the main changes are to explicitly parameterize target hierarchies, and to replace concrete embedded functions/relations with abstract ones satisfying reasonable conditions.

To begin with, we define basic terms and symbols. Let $\boldsymbol{X}$ be the set of variables, $\boldsymbol{D}$ the domain of the variables, and $\boldsymbol{C}$ the set of constraints. Given a constraint c, $\boldsymbol{X}(c)$ denotes the set of all the variables constrained by c, and given a set C of constraints, we define

$$\boldsymbol{X}(C) = \{x \in \boldsymbol{X} \mid \exists c \in C.\, x \in \boldsymbol{X}(c)\}$$

which consists of all the variables constrained by some constraints in C. A strength is an integer between 0 and l_{w}, where l_{w} is some positive integer. Intuitively, the larger the integer is, the weaker the strength is. Let $\boldsymbol{L}$ be the set of all the strengths. A labeled constraint c/l is a constraint c associated with a strength l. A constraint hierarchy is a finite multi-set H of labeled constraints[4], and $\boldsymbol{H}$ expresses the set of all constraint hierarchies. For convenience, we define H/l as follows:[5]

$$H/l \equiv \{c \in \boldsymbol{C} \mid c/l \in H\}.$$

Shortly, H/l represents the set of all constraints in level l of H. Note that the strength l is detached from the labeled constraints.

To represent solutions of constraint hierarchies, we also use variable assignments. A variable assignment, denoted as θ, is a mapping from $\boldsymbol{X}$ to $\boldsymbol{D}$, and $\boldsymbol{\Theta}$ indicates the set of all variable assignments. Given a set X of variables, we define $\theta(X) = \theta'(X)$ as follows:

$$\theta(X) = \theta'(X) \equiv \forall x \in X.\, \theta(x) = \theta'(x).$$

That is, with θ and θ', all variables in X have equal values.

To assign semantics to constraints, we first introduce error functions in the same way as the original formulation:

[4]Since constraint hierarchies are multi-sets, they can contain multiple copies of a labeled constraint. In the following discussion, we assume that set operations for constraint hierarchies are the ones for multi-sets. For example, merging two constraint hierarchies H and H', both of which contain one copy of a labeled constraint c/l, the resulting hierarchy $H \cup H'$ has two copies of c/l.

[5]We assume that when H has multiple copies of c/l, H/l also contains the same number of copies of c.

Definition 2.1 (error function). An error function for level l is a mapping $e_l : \boldsymbol{C} \times \boldsymbol{\Theta} \to \{0\} \cup \boldsymbol{R}^+$ such that for any constraint c and variable assignments θ and θ',

$$\theta(\boldsymbol{X}(c)) = \theta'(\boldsymbol{X}(c)) \Rightarrow e_l(c, \theta) = e_l(c, \theta').$$

Intuitively, $e_l(c, \theta)$ indicates the error of a labeled constraint c/l for θ, which is zero if c/l is exactly satisfied, and positive otherwise. The condition requires that errors of a constraint for two variable assignments are equal if the assignments have equal values for each constrained variable.[6]

Next, we introduce *level comparators*:

Definition 2.2 (level comparator). A level comparator for level l is a ternary relation $\overset{\cdot/l}{\lesssim}: \boldsymbol{H} \times \boldsymbol{\Theta} \times \boldsymbol{\Theta}$ such that for any constraint hierarchies H and H' and variable assignments θ, θ', and θ'',

$$H/l = H'/l \Rightarrow (\theta \overset{H/l}{\lesssim} \theta' \Leftrightarrow \theta \overset{H'/l}{\lesssim} \theta') \tag{2.3}$$

$$\forall c \in H/l.\, e_l(c, \theta) = e_l(c, \theta'') \Rightarrow (\theta \overset{H/l}{\lesssim} \theta' \Leftrightarrow \theta'' \overset{H/l}{\lesssim} \theta') \tag{2.4}$$

$$\forall c \in H/l.\, e_l(c, \theta') = e_l(c, \theta'') \Rightarrow (\theta \overset{H/l}{\lesssim} \theta' \Leftrightarrow \theta \overset{H/l}{\lesssim} \theta'') \tag{2.5}$$

$$\forall c \in H/l.\, e_l(c, \theta) \leq e_l(c, \theta') \Rightarrow \theta \overset{H/l}{\lesssim} \theta' \tag{2.6}$$

$$\theta \overset{H/l}{\lesssim} \theta' \wedge \theta' \overset{H/l}{\lesssim} \theta'' \Rightarrow \theta \overset{H/l}{\lesssim} \theta'' \tag{2.7}$$

$$\theta \overset{H/l}{\lesssim} \theta' \wedge \theta \overset{H'/l}{\lesssim} \theta' \Rightarrow \theta \overset{(H \cup H')/l}{\lesssim} \theta' \tag{2.8}$$

$$(\forall c \in H/l.\, e_l(c, \theta) = 0) \wedge (\exists c \in H/l.\, e_l(c, \theta') > 0) \Rightarrow \theta \overset{H/l}{\lesssim} \theta' \wedge \neg\, \theta' \overset{H/l}{\lesssim} \theta. \tag{2.9}$$

Intuitively, $\theta \overset{H/l}{\lesssim} \theta'$ means "θ is better than or similar to θ' according to l of H." Conditions (2.3)–(2.5) say that the scope of a level comparator is restricted to be inside a designated level. Condition (2.6) indicates that if errors of all constraints at a level for an assignment are smaller than or equal to those for another assignment, then the former assignment is better than or similar to the latter according to the level. Condition (2.7) is 'transitivity'

[6] This property is quite natural as a requirement. However, the original theory does not dictate the property although it may assume it implicitly; it does not need the property as far as it discusses. By contrast, we require it to further investigate constraint hierarchies.

of a level comparator. Condition (2.8) means that if, in two hierarchies, an assignment is better than or similar to another according to the level, then the relation holds in the combination of the hierarchies. Condition (2.9) corresponds to 'respecting every hierarchies,' which is presented in [94] as a convincing sufficient condition for the property '*better* respects the hierarchy.'

For convenience, we define $\overset{\cdot/l}{\gtrsim}$ (worse than or similar to), $\overset{\cdot/l}{\sim}$ (similar to), $\overset{\cdot/l}{<}$ (better than), $\overset{\cdot/l}{>}$ (worse than), and $\overset{\cdot/l}{\not\sim}$ (incomparable with) as follows:

$$\begin{aligned}
\theta \overset{H/l}{\gtrsim} \theta' &\Leftrightarrow \theta' \overset{H/l}{\lesssim} \theta \\
\theta \overset{H/l}{\sim} \theta' &\Leftrightarrow \theta \overset{H/l}{\lesssim} \theta' \wedge \theta \overset{H/l}{\gtrsim} \theta' \\
\theta \overset{H/l}{<} \theta' &\Leftrightarrow \theta \overset{H/l}{\lesssim} \theta' \wedge \neg\, \theta \overset{H/l}{\sim} \theta' \\
\theta \overset{H/l}{>} \theta' &\Leftrightarrow \theta \overset{H/l}{\gtrsim} \theta' \wedge \neg\, \theta \overset{H/l}{\sim} \theta' \\
\theta \overset{H/l}{\not\sim} \theta' &\Leftrightarrow \neg\, \theta \overset{H/l}{\lesssim} \theta' \wedge \neg\, \theta \overset{H/l}{\gtrsim} \theta'.
\end{aligned}$$

Level comparators correspond to $<_g$ and $<>_g$ in the original formulation. The differences are briefly summarized as follows: the original formulation separates $\overset{\cdot/l}{\lesssim}$ into $<_g$ and $<>_g$, and defines them constructively; it includes (2.3)–(2.5) operationally; it seems to implicitly assume (2.6); it does not require the transitivity of $\overset{\cdot/l}{\sim}$ unlike (2.7) (in spite of assuming it for most instances); it presents no condition like (2.8).

We can define level comparators in the same way as the original theory. For example, the *least-squares level comparator* is defined as follows:[7]

$$\theta \overset{H/l}{\lesssim} \theta' \Leftrightarrow \sum_{c/l \in H} e_l(c,\theta)^2 \leq \sum_{c/l \in H} e_l(c,\theta')^2.$$

Here, two variable assignments are compared by summing squares of errors of constraints at the level. It is easy to prove that the definition fulfills the conditions for level comparators. Used in satisfaction of constraint hierarchies, it works as the least-squares method within level l.

[7] While the original theory provides the explicit use of weights of constraints in least-squares-better, we omit them for simplicity. However, even if we incorporate weights, we can obtain the similar results presented in this research.

Next, we define *hierarchy comparators* that totally consider constraint hierarchies by combining level comparators:

Definition 2.3 (hierarchy comparator). A hierarchy comparator is a ternary relation $\dot{<} : \boldsymbol{H} \times \boldsymbol{\Theta} \times \boldsymbol{\Theta}$ such that for any constraint hierarchy H and variable assignments θ and θ',

$$\theta \overset{H}{<} \theta' \Leftrightarrow \exists l \in \boldsymbol{L}.\, (\forall l' \in \boldsymbol{L}.\, l' < l \Rightarrow \theta \overset{H/l'}{\sim} \theta') \wedge \theta \overset{H/l}{<} \theta'.$$

Intuitively, $\theta \overset{H}{<} \theta'$ means "θ is better than θ' according to H." It is defined as lexicographic ordering with level comparators as its components. Consequently, the result of a level comparator has absolute priority over those of weaker ones.

For convenience, we define $\dot{>}$ (worse than), $\dot{\sim}$ (similar to), $\dot{\lesssim}$ (better than or similar to), $\dot{\gtrsim}$ (worse than or similar to), and $\dot{\not\sim}$ (incomparable with) as follows:

$$\begin{aligned}
\theta \overset{H}{>} \theta' &\Leftrightarrow \theta' \overset{H}{<} \theta \\
\theta \overset{H}{\sim} \theta' &\Leftrightarrow \forall l \in \boldsymbol{L}.\, \theta \overset{H/l}{\sim} \theta' \\
\theta \overset{H}{\lesssim} \theta' &\Leftrightarrow \theta \overset{H}{<} \theta' \vee \theta \overset{H}{\sim} \theta' \\
\theta \overset{H}{\gtrsim} \theta' &\Leftrightarrow \theta \overset{H}{>} \theta' \vee \theta \overset{H}{\sim} \theta' \\
\theta \overset{H}{\not\sim} \theta' &\Leftrightarrow \neg\, \theta \overset{H}{\lesssim} \theta' \wedge \neg\, \theta \overset{H}{\gtrsim} \theta'.
\end{aligned}$$

The following definition describes satisfaction of constraint hierarchies using a hierarchy comparator:

Definition 2.4 (constraint hierarchy satisfier). A constraint hierarchy satisfier is a mapping $S : 2^{\boldsymbol{\Theta}} \times \boldsymbol{H} \to 2^{\boldsymbol{\Theta}}$ defined as

$$S(\Theta, H) = \{\theta \in \Theta \mid \neg\, \exists \theta' \in \Theta.\, \theta' \overset{H}{<} \theta\}.$$

As a shorthand, we write $S(H)$ instead of $S(\boldsymbol{\Theta}, H)$. Intuitively, $S(\Theta, H)$ is the set of assignments obtained by satisfying H using assignments in Θ. By definition, an assignment in $S(\Theta, H)$ is an element in Θ such that there is no better assignment in Θ when compared according to H.

Finally, we define solutions of constraint hierarchies:

Definition 2.5 (solution). A solution to H is a variable assignment in $S(H)$.

In other words, a solution of H is an assignment found by satisfying H in the set of all assignments.

Hierarchy comparators are simply called 'comparator' in the original formulation, that is, $\dot{<}$ corresponds to $<_G$. One difference is that the original formulation restricts a single kind of level comparators in defining a hierarchy comparator. By contrast, our formalization allows us to combine multiple kinds of level comparators in a hierarchy comparator; such combination is sometimes useful for practical purposes.

Another difference is that the original formulation restricts top-level constraints to be required, whereas ours allows conflicting constraints at the top level. This is because our definition of hierarchy satisfiers excludes the special treatment of the top level. However, the resulting solutions are the same so far as the top level is not over-constrained. Also, even if we add the condition for the top level to be required, we can accommodate it in our following proofs (but they should be rather complicated).

2.2.2 Global Semi-Monotonicity

We define a useful property called *global semi-monotonicity* (GSM) in satisfying constraint hierarchies as follows:

Definition 2.6 (global semi-monotonicity). S is globally semi-monotonic iff for any H and H',

$$S(H) \cap S(H') \subseteq S(H \cup H').$$

GSM requires that any common solution of two constraint hierarchies is also a solution of their combination. It is not only natural but also weak (or general) in a sense that the condition is true for any two hierarchies sharing no solutions.

GSM, by definition, is not limited to constraint hierarchies. In a similar style, we can express basic properties of various constraint systems. For example, we can represent ordinary monotonicity as

$$S(H) \cap S(H') = S(H \cup H')$$

where the difference from GSM is that it has

$$S(H) \cap S(H') \supseteq S(H \cup H').$$

Thus we can see that GSM lacks the familiar style of the monotonic property,

$$S(H) \supseteq S(H \cup H')$$

which means that adding constraints to a constraint system either preserves or reduces their solutions.[8]

We present a useful class of GSM constraint hierarchy satisfiers called *global constraint hierarchy satisfiers*, using *global level comparators* and *global hierarchy comparators*:

Definition 2.7 (global level comparator). A level comparator $\overset{\cdot/l}{\lesssim}$ is global iff for any constraint hierarchies H and H' and variable assignments θ and θ',

$$\theta \overset{H/l}{<} \theta' \wedge \theta \overset{H'/l}{\sim} \theta' \;\Rightarrow\; \theta \overset{(H\cup H')/l}{<} \theta' \tag{2.10}$$

$$\theta \overset{H/l}{\not\sim} \theta' \;\Rightarrow\; \neg\,\theta \overset{(H\cup H')/l}{\sim} \theta' \tag{2.11}$$

$$\neg\,\theta \overset{H/l}{<} \theta' \wedge \neg\,\theta \overset{H'/l}{<} \theta' \;\Rightarrow\; \neg\,\theta \overset{(H\cup H')/l}{<} \theta'. \tag{2.12}$$

Definition 2.8 (global hierarchy comparator). A hierarchy comparator is global iff each of its level comparators is global.

Definition 2.9 (global constraint hierarchy satisfier). A constraint hierarchy satisfier is global iff its hierarchy comparator is global.

An example of global level comparators is the least-squares level comparator. Most level comparators presented in the original formulation are also global, which we will discuss later.

The following theorem proves that global constraint hierarchy satisfiers are GSM:

Theorem 2.10. *Let S be an arbitrary global constraint hierarchy satisfier. Then S is GSM.*

Proof. By contradiction: Assume that S is global, and that for some constraint hierarchies H and H',

$$S(H) \cap S(H') \subseteq S(H \cup H')$$

does not hold, that is, there exists a variable assignment θ that is in $S(H)$ and $S(H')$, but not in $S(H \cup H')$. Then, for some assignment θ', $\theta' \overset{H\cup H'}{<} \theta$ holds, that is, for some level l,

$$(\forall l' \in \boldsymbol{L}.\, l' < l \Rightarrow \theta' \overset{(H\cup H')/l'}{\sim} \theta) \wedge \theta' \overset{(H\cup H')/l}{<} \theta.$$

[8] We believe that such a universal style of formal properties is helpful in comparing different nonmonotonic systems.

By (2.10) and (2.11), $\theta' \overset{(H\cup H')/l'}{\sim} \theta$ implies

$$(\theta' \overset{H/l'}{<} \theta \wedge \theta' \overset{H'/l'}{>} \theta) \vee (\theta' \overset{H/l'}{\sim} \theta \wedge \theta' \overset{H'/l'}{\sim} \theta) \vee (\theta' \overset{H/l'}{>} \theta \wedge \theta' \overset{H'/l'}{<} \theta)$$

and by (2.12), $\theta' \overset{(H\cup H')/l}{<} \theta$ implies

$$\theta' \overset{H/l}{<} \theta \vee \theta' \overset{H'/l}{<} \theta.$$

Hence, at least one of the following two cases must hold:

1. *Case*

   $$\exists l' \in \boldsymbol{L}.\, l' \leq l \wedge (\forall l'' \in \boldsymbol{L}.\, l'' < l' \Rightarrow \theta' \overset{H/l''}{\sim} \theta \wedge \theta' \overset{H'/l''}{\sim} \theta) \wedge \theta' \overset{H/l'}{<} \theta.$$

   Then $\theta' \overset{H}{<} \theta$ holds, which is a contradiction to $\theta \in S(H)$.

2. *Case*

   $$\exists l' \in \boldsymbol{L}.\, l' \leq l \wedge (\forall l'' \in \boldsymbol{L}.\, l'' < l' \Rightarrow \theta' \overset{H/l''}{\sim} \theta \wedge \theta' \overset{H'/l''}{\sim} \theta) \wedge \theta' \overset{H'/l'}{<} \theta.$$

   Then $\theta' \overset{H'}{<} \theta$ holds, which is a contradiction to $\theta \in S(H')$.

Both of the two cases resulted in contradiction. Thus, for any constraint hierarchies H and H',

$$S(H) \cap S(H') \subseteq S(H \cup H')$$

holds. Therefore, S is GSM. □

The converse, that GSM satisfiers are global, is not true; in fact, we have not found weaker conditions for level comparators that yield a set equivalent to GSM. However, we believe that most useful GSM satisfiers are global.[9]

[9] Actually, we could make the converse true if we strengthened the formulation of constraint hierarchies by allowing only 'modular' hierarchy comparators as follows: let level comparators be in a certain set including the least-squares level comparator, and also let hierarchy comparators need to be arbitrarily composed of level comparators in the set. For modular hierarchy comparators, the truth of the converse is easily provable since we can create a non-GSM satisfier by combining any non-global and the least-squares level comparators. Another set of level comparators without the least-squares level comparator may exist, but is unlikely to be more useful.

Global hierarchy comparators might seem strongly related to globally-better comparators in the original formulation, but in fact, they are different. One instance of globally-better, least-squares-better, is composed of the least-squares level comparators, and therefore, is global. However, worst-case-better is not global because (2.10) does not hold. Generally, for level comparators for globally-better, (2.11) is true since they compare real numbers, i.e. $\neg\, \theta \overset{H/l}{\not\sim} \theta'$. However, it depends on actual instances of level comparators whether both (2.10) and (2.12) hold.

It should also be noted that even locally-better comparators are global. Before showing it, we define *local level comparators*, *local hierarchy comparators*, and *local constraint hierarchy satisfiers*:

Definition 2.11 (local level comparator). A level comparator $\overset{\cdot/l}{\lesssim}$ is local iff for any constraint hierarchy H and variable assignments θ and θ',

$$\theta \overset{H/l}{\lesssim} \theta' \Rightarrow \forall c \in H/l.\, e_l(c,\theta) \leq e_l(c,\theta'). \tag{2.13}$$

Definition 2.12 (local hierarchy comparator). A hierarchy comparator is local iff each of its level comparators is local.

Definition 2.13 (local constraint hierarchy satisfier). A constraint hierarchy satisfier is local iff its hierarchy comparator is local.

By (2.6) and (2.13), a local level comparator results in

$$\theta \overset{H/l}{\lesssim} \theta' \Leftrightarrow \forall c \in H/l.\, e_l(c,\theta) \leq e_l(c,\theta').$$

Because of the definitions of $\overset{\cdot l}{\sim}$ and $\overset{\cdot l}{<}$, we can obtain the following relations:

$$\begin{aligned}
\theta \overset{H/l}{\sim} \theta' &\Leftrightarrow \forall c \in H/l.\, e_l(c,\theta) = e_l(c,\theta') \\
\theta \overset{H/l}{<} \theta' &\Leftrightarrow (\forall c \in H/l.\, e_l(c,\theta) \leq e_l(c,\theta')) \wedge \\
&\quad\ (\exists c' \in H/l.\, e_l(c',\theta) < e_l(c',\theta'))
\end{aligned}$$

which are equivalent to (2.1) and (2.2) for defining locally-better comparators in the original formalization.

Now, we can prove the following proposition:

Proposition 2.14. *Any local level comparator is global.*
Proof. Straightforward by Definitions 2.7. □

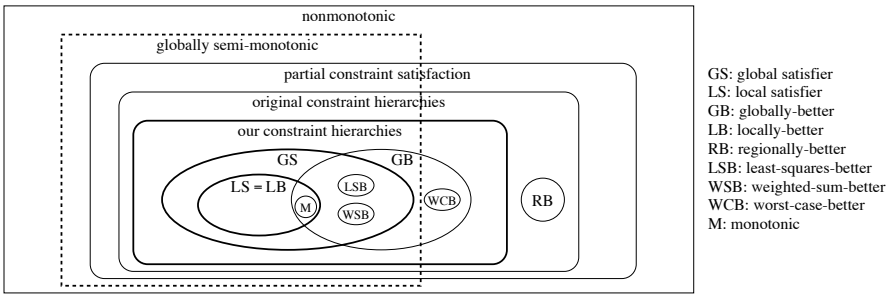

Figure 2.1: Relationship of nonmonotonic constraint systems.

Clearly, it follows that local hierarchy comparators and local constraint hierarchy satisfiers are also global.

The proposition provides a critical difference between global hierarchy and globally-better comparators: although original formulation presented locally-better and globally-better as separate concepts, we successfully integrated locally-better and an important class of globally-better into global hierarchy comparators via GSM.

2.3 Discussion

In this section, we review the relationship between the original and our theories of constraint hierarchies, which is roughly illustrated in Figure 2.1.

Our reformulation of constraint hierarchies has become narrower than the original one,[10] because we necessitated $\overset{\cdot/l}{\sim}$ to be transitive by (2.7). For example, we exclude regionally-better in [90] since for its $\overset{H/l}{\sim}$ is not transitive. However, excluding such level comparators contributed to theoretical cleanness.

Except regionally-better and worst-case-better, all the hierarchy comparators presented in the original formulation are global by our formulation. We believe that this fact supports the expressiveness of our global comparators with respect to constraint hierarchies. Furthermore, in Chapter 3, we show that global comparators are useful in designing efficient constraint satisfaction algorithms.

[10] Strictly speaking, as noted earlier, our theory allows conflicting constraints at the top level, while the original theory restricts top-level constraints to be required.

Chapter 3

Generalized Local Propagation

This chapter proposes generalized local propagation (GLP) as a theoretical framework for scheduling constraints in constraint hierarchies.

3.1 Motivation

Local propagation takes advantage of the potential locality of typical (possibly, non-hierarchical) constraint systems in graphical user interfaces. Basically, it is efficient because it uniquely solves a single constraint in each step. In addition, when a variable value is repeatedly updated by an operation such as dragging in interactive interfaces, it can easily re-evaluate only the necessary constraints using the same schedule or plan. Furthermore, most local propagation algorithms improve their efficiency by incrementally updating plans for solving constraint systems when constraints are added or removed.

However, local propagation has been restricted to locally-better comparators (i.e. local hierarchy comparators) and *multi-way constraints*, which can be uniquely solved for each variable, e.g. linear equations over real numbers.[1] Also, it cannot find multiple solutions for a given constraint hierarchy due to the uniqueness.

Naturally, a question arises whether we can 'generalize' local propagation to solve hierarchies of more powerful constraints without losing its efficiency.

[1] Multi-way constraints are similar to functional constraints, which were introduced in the AC-5 arc-consistency algorithm [32]. However, their contexts are quite different, because arc consistency targets binary constraint satisfaction problems over finite domains.

In this chapter, we propose *generalized local propagation*, a theoretical framework for investigating local propagation on constraint hierarchies, and show that global semi-monotonicity presented in Chapter 2 exhibits a practically useful property in generalized local propagation.

3.2 The Theory of Generalized Local Propagation

This section provides the theory of generalized local propagation.

3.2.1 Formulation

Classical local propagation satisfies a constraint system by successively solving individual constraints in an order closely associated with the network topology of the system. Here we generalize local propagation so that it can solve a set of constraints in one step and also adopt an arbitrary order among such constraint sets. For this purpose, we introduce *ordered partitions* as follows:

Definition 3.1 (ordered partition). A *partition* of a constraint hierarchy is a set P generated by decomposing the hierarchy into disjoint subsets called *blocks*. An ordered partition of P is a pair $\langle P, \leq_P \rangle$, where $\leq_P$ is an arbitrary partial order among blocks in P.

Obviously, the original constraint hierarchy of P is the combination of all blocks in P, i.e.

$$H = \bigcup_{B \in P} B.$$

For brevity, we write $B <_P B'$ instead of $B \leq_P B' \wedge B \neq B'$ for blocks B and B' in P.

Ordered partitions are easily illustrated with diagrams. For instance, consider the ordered partition $\langle P, \leq_P \rangle$ consisting of the blocks B_1, B_2, ..., B_9, as illustrated in Figure 3.1. The partial order $\leq_P$ is defined as the reflexive transitive closure of all the arrows in Figure 3.1.

Using ordered partitions into blocks, we define *generalized local propagation* (GLP) in the following way:

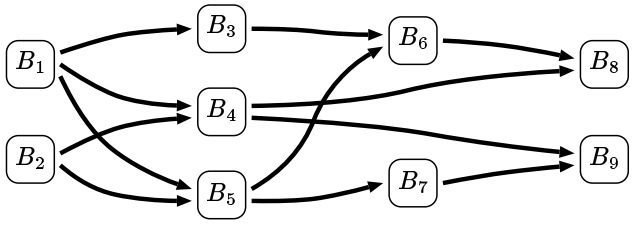

Figure 3.1: An ordered partition.

Definition 3.2 (generalized local propagation). Generalized local propagation with S is a mapping $\pi_S(\langle P, \leq_P \rangle)$ defined as follows:

$$\pi_S(\langle P, \leq_P \rangle) = \begin{cases} \boldsymbol{\Theta} & \text{if } |P| = 0 \\ \bigcap_{B \in \mathit{terminals}(\langle P, \leq_P \rangle)} S(\pi_S(\mathit{before}(B, \langle P, \leq_P \rangle)), B) & \text{otherwise,} \end{cases}$$

where *terminals* and *before* are as follows:

$$\begin{aligned} \mathit{terminals}(\langle P, \leq_P \rangle) &= \{B' \in P \mid \neg\, \exists\, B'' \in P.\, B' <_P B''\} \\ \mathit{before}(B, \langle P, \leq_P \rangle) &= \langle P', \leq_{P'} \rangle \end{aligned}$$

where

$$\begin{aligned} P' &= \{B' \in P \mid B' <_P B\} \\ \leq_{P'} &= \{\langle B', B'' \rangle \in P' \times P' \mid B' \leq_P B''\}. \end{aligned}$$

Intuitively, $\mathit{terminals}(\langle P, \leq_P \rangle)$ is the set of all blocks at terminal positions, and $\mathit{before}(B, \langle P, \leq_P \rangle)$ is an 'ordered sub-partition' of $\langle P, \leq_P \rangle$ where all blocks are before B. For example, reconsider the ordered partition $\langle P, \leq_P \rangle$ in Figure 3.1. Here $\mathit{terminals}(\langle P, \leq_P \rangle)$ is the set $\{B_8, B_9\}$ as shown in Figure 3.2. Also, $\mathit{before}(B_9, \langle P, \leq_P \rangle)$ is the pair of the set $\{B_1, B_2, B_4, B_5, B_7\}$ and the partial order defined as the reflexive transitive closure of the black arrows. Thus, B_9 is satisfied in the set of assignments obtained by applying GLP to blocks before B_9. Accordingly, we can view GLP as a process that successively solves each blocks in some order respecting $\leq_P$. This is always possible because $\leq_P$ is a partial order.

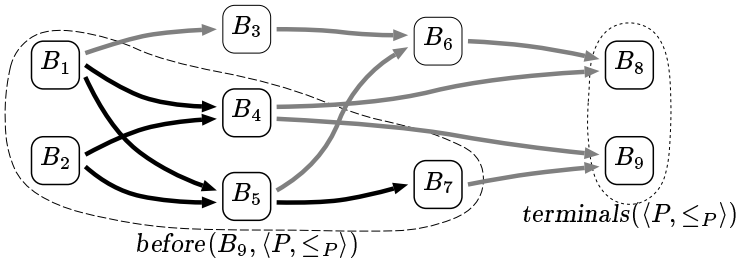

Figure 3.2: Generalized local propagation.

3.2.2 Properties of Global Hierarchy Comparators

In this subsection, focusing on global hierarchy comparators, we show useful properties for finding solutions of constraint hierarchies with GLP.

As a first step, we prove the next lemma, which means that by using a global hierarchy comparator, GLP respects the similarity of variable assignments for ordered partitions that satisfy the conditions below:

Lemma 3.3. *Let S be an arbitrary global constraint hierarchy satisfier. Also, let H be an arbitrary constraint hierarchy, $\langle P, \leq_P \rangle$ an arbitrary ordered partition of H, and θ an arbitrary variable assignment in $\pi_S(\langle P, \leq_P \rangle)$. Then, for any assignment θ', if $\theta' \overset{H}{\sim} \theta$ and*

$$\begin{aligned} (3.1) \qquad & \forall B \in P. \forall c/l \in B. e_l(c, \theta) > 0 \\ & \Rightarrow \quad (\forall B' \in P. B' <_P B \Rightarrow \forall c'/l' \in B'. l' < l) \end{aligned}$$

then θ' is in $\pi_S(\langle P, \leq_P \rangle)$.

Proof. By contradiction: Assume that there exists some assignment θ' which is not in $\pi_S(\langle P, \leq_P \rangle)$. Then it is necessary that for some block B_1 in P, θ' is in $\pi_S(before(B_1, \langle P, \leq_P \rangle))$, but not in $S(\pi_S(before(B_1, \langle P, \leq_P \rangle)), B_1)$. Because $\theta' \overset{H}{\sim} \theta$ holds and S is global, $\theta \overset{B_1}{\not\sim} \theta'$ does not hold. Therefore, $\theta \overset{B_1}{<} \theta'$ must hold, that is, there exists some level l_1 such that

$$(\forall l \in \boldsymbol{L}. l < l_1 \Rightarrow \theta \overset{B_1/l}{\sim} \theta') \wedge \theta \overset{B_1/l_1}{<} \theta'.$$

This implies that for some block B in P, $\theta \overset{B/l_1}{>} \theta'$ holds. Since θ must be in $S(\pi_S(before(B, \langle P, \leq_P \rangle)), B)$, at least one of the following two cases must hold:

1. *Case*

$$\theta' \in \pi_S(\mathit{before}(B, \langle P, \leq_P \rangle)) \wedge \theta' \notin S(\pi_S(\mathit{before}(B, \langle P, \leq_P \rangle)), B).$$

Since $\theta \overset{B/l_1}{>} \theta'$ holds, there must exist some level l_2 such that $l_2 < l_1$ and $\theta \overset{B/l_2}{<} \theta'$.

2. *Case*

$$\theta' \notin \pi_S(\mathit{before}(B, \langle P, \leq_P \rangle)).$$

Then, for some block B' in P such that $B' <_P B$, θ' is in $\pi_S(\mathit{before}(B', \langle P, \leq_P \rangle))$, but not in $S(\pi_S(\mathit{before}(B', \langle P, \leq_P \rangle)), B')$. Since $\theta \overset{B/l_1}{>} \theta'$ implies

$$\exists c \in B/l_1 . e_{l_1}(c, \theta) > 0$$

and also since (3.1) holds, B' contains only stronger constraints than l_1. Therefore, there exists some level l_2 such that $l_2 < l_1$ and $\theta \overset{B'/l_2}{<} \theta'$.

Beginning with $\theta \overset{B_1/l_1}{<} \theta'$, both of the two cases resulted in that there exist some level l_2 and block B_2 such that $l_2 < l_1$ and $\theta \overset{B_2/l_2}{<} \theta'$. Clearly, it causes an infinite sequence $l_1, l_2, \ldots$ such that $l_i > l_{i+1}$. However, since each l_i is a non-negative integer, it is a contradiction. □

Intuitively, Lemma 3.3 says that if GLP using a global satisfier generates a variable assignment for which constraints with errors have only stronger constraints before them, then it yields all similar (i.e. $\overset{H}{\sim}$) assignments. Note that the sufficient condition (3.1) allows constraints without errors to be placed after weaker ones.

In the following theorem, we prove that such variable assignments are solutions of the constraint hierarchy:

Theorem 3.4. *Let S be an arbitrary global constraint hierarchy satisfier. Then, for any constraint hierarchy H, ordered partition $\langle P, \leq_P \rangle$ of H, and variable assignment θ in $\pi_S(\langle P, \leq_P \rangle)$, θ is a solution of H if (3.1) holds.*

Proof. By induction on the size of P:

Induction base: If $|P| = 0$, the proposition holds.

Induction step: Assume that if $|P| < n$, the proposition holds. Now, let $|P| = n$. For any block B in $\mathit{terminals}(\langle P, \leq_P \rangle)$, θ must be in

$S(\pi_S(before(B, \langle P, \leq_P \rangle)), B)$. Therefore, by the induction hypothesis, θ is in $S(H_B)$, where H_B is the union of blocks of $before(B, \langle P, \leq_P \rangle)$. Now, we assume (for contradiction) that there exists some assignment θ' such that $\theta' \overset{H_B \cup B}{<} \theta$, that is, for some level l,

$$(\forall l' \in \boldsymbol{L}. l' < l \Rightarrow \theta' \overset{(H_B \cup B)/l'}{\sim} \theta) \wedge \theta' \overset{(H_B \cup B)/l}{<} \theta.$$

Then at least one of the following two cases must hold:

1. *Case*

   $$\exists l' \in \boldsymbol{L}. l' \leq l \wedge (\forall l'' \in \boldsymbol{L}. l'' < l' \Rightarrow \theta' \overset{H_B/l''}{\sim} \theta \wedge \theta' \overset{B/l''}{\sim} \theta) \wedge \theta' \overset{H_B/l'}{<} \theta.$$

   Then $\theta' \overset{H_B}{<} \theta$ holds. Therefore, $\theta \notin S(H_B)$, which is a contradiction.

2. *Case*

   $$\exists l' \in \boldsymbol{L}. l' \leq l \wedge (\forall l'' \in \boldsymbol{L}. l'' < l' \Rightarrow \theta' \overset{H_B/l''}{\sim} \theta \wedge \theta' \overset{B/l''}{\sim} \theta) \wedge \theta' \overset{B/l'}{<} \theta.$$

   Then, for some labeled constraint c/l' in B, $e_{l'}(c, \theta) > 0$ must hold. By (3.1), H_B contains only stronger constraints than l'. Therefore, $\theta' \overset{H_B}{\sim} \theta$ holds. By Lemma 3.3, θ' is also in $\pi_S(before(B, \langle P, \leq_P \rangle))$. However, since $\theta' \overset{B}{<} \theta$ holds, it implies $\theta \notin S(\pi_S(before(B, \langle P, \leq_P \rangle)), B)$, which is a contradiction.

Both cases caused contradiction. Therefore, there never exists such θ', i.e. θ is in $S(H_B \cup B)$. Since S is global, θ is also in $S(H)$ by Theorem 2.10. □

The theorem presents a strategy to design algorithms for solving constraint hierarchies. As noted, the sufficient condition permits constraints without errors to be located after weaker ones. In other words, we can delay satisfaction of a strong constraint with no error until some appropriate time, for example, "when the constraint becomes uniquely satisfiable."

An important instance of such GLP is the refining method. Since constraints have no weaker constraints before them in the method, it can be easily understood by Theorem 3.4 that it generates only correct solutions, i.e. is sound. In addition, using a certain kind of global hierarchy comparators, the refining method yields all solutions, i.e. is complete:

Proposition 3.5. *Let S be an arbitrary global constraint hierarchy satisfier such that for any constraint hierarchy H and variable assignments θ and θ', $\neg\, \theta \overset{H}{\not\sim} \theta'$. Then, for any constraint hierarchy H, the following holds:*

$$\pi_S(\langle P, \leq_P \rangle) = S(H)$$

where

$$\begin{aligned} P &= \{B_l \mid l \in \boldsymbol{L}\} \\ \leq_P &= \{\langle B_l, B_{l'} \rangle \in P \times P \mid l \leq l'\} \end{aligned}$$

where B_l is defined as follows:[2]

$$B_l \equiv \{c/l' \in H \mid l' = l\}.$$

Proof. Straightforward by Theorem 3.4. □

By this proposition, a refining-method algorithm using a global hierarchy and globally-better comparator, e.g. least-squares-better, is sound and complete, because any globally-better comparator satisfies $\neg\, \theta \overset{H}{\not\sim} \theta'$.[3]

3.2.3 Properties of Local Hierarchy Comparators

Using a local hierarchy comparator, we can obtain a theorem with a weaker sufficient condition than that of Theorem 3.4:

Theorem 3.6. *Let S be an arbitrary local constraint hierarchy satisfier. Then, for any constraint hierarchy H, ordered partition $\langle P, \leq_P \rangle$ of H, and variable assignment θ in $\pi_S(\langle P, \leq_P \rangle)$, θ is a solution of H if*

$$\begin{aligned} &\forall B \in P.\, \forall c/l \in B.\, e_l(c, \theta) > 0 \\ &\quad \Rightarrow \quad (\forall B' \in P.\, B' <_P B \Rightarrow \forall c'/l' \in B'.\, l' \leq l). \end{aligned} \tag{3.2}$$

Proof. Similar to that of Theorem 3.4. □

The difference of (3.2) from (3.1) is the existence of equality in $l' \leq l$, which indicates that (3.2) is weaker than (3.1). Since it will provide more freedom to organize ordered partitions or schedule constraints, we can expect to develop more efficient constraint solving algorithms using local comparators.

[2] B_l is not H/l; B_l contains labeled constraints while H/l is a set of constraints.

[3] It is probably possible to weaken the sufficient conditions for level comparators, because ordered partitions for the refining method are too special.

3.3 Relationship with the DeltaBlue Algorithm

The results that we obtained in GLP gracefully explain why past local propagation algorithms solve constraint hierarchies, which we believe is an evidence of the usefulness of GLP as a theoretical basis for solving constraint hierarchies. In this section, we relate GLP with the DeltaBlue algorithm [22, 58, 76], which is one of the most famous algorithms that maintain constraint hierarchies with local propagation.

DeltaBlue expresses constraint hierarchies as *constraint graphs*, which are bipartite graphs with variables and constraints as nodes.[4] In DeltaBlue, constraints are *multi-way*. Intuitively, a multi-way constraint can be uniquely solved for any one of its variables. For example, the constraint

$$x = y + z$$

is multi-way because it can be solved for x, y, and z as follows:

$$x \leftarrow y + z$$
$$y \leftarrow x - z$$
$$z \leftarrow x - y.$$

DeltaBlue refers such an expression for computing a variable value as a *method*, and represents each constraint as a set of methods.

The task of DeltaBlue as a local propagation algorithm is, in short, to determine which constraints in a hierarchy to satisfy, which method to select for satisfying each constraint, and in what order to compute selected methods. To guarantee that such a set of selected methods generates a correct solution, it introduces concepts known as *walkabout strengths*. In DeltaBlue, walkabout strengths, associated with variables, are defined to propagate strengths of the weakest constraints:

Definition 3.7 (walkabout strength [22]). Variable x is determined by method m of constraint c. x's walkabout strength is the weakest of c's strength and the walkabout strengths of m's input.

To see how to compute a walkabout strength, let us quote the example from [22] as illustrated in Figure 3.3. Here the squares represent constraints, and the circles indicate variables. Also, the edges mean that variables are constrained by constraints connected with the edges, and the directed edges

[4] In the papers on DeltaBlue [22, 58, 76], they do not associate constraint hierarchies with bipartite graphs. However, such a connection is often convenient to explain a certain kind of local propagation algorithms [25].

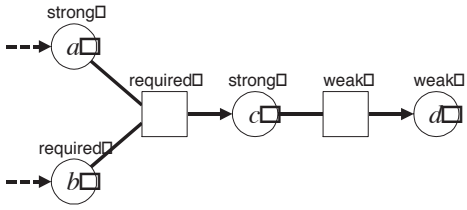

Figure 3.3: Walkabout strengths.

show selected methods by pointing output variables. Symbols associated with constraints give their strengths, and ones attached to variables depict their walkabout strengths. In this example, the walkabout strength of variable c is strong because the constraint determining c is required and the walkabout strengths of the inputs a and b to the constraint are strong and required respectively. Thus walkabout strengths propagate strengths of the weakest constraints along selected methods.[5]

With walkabout strengths, a sufficient condition for obtaining correct solutions of constraint hierarchies can be described, which is called '*blocked constraint lemma*:'

Definition 3.8 (blocked constraint [22]). A blocked constraint is an unsatisfied constraint whose strength is stronger than the walkabout strength of one of its potential output variables.

Lemma 3.9 (blocked constraint lemma [22]). *If there are no blocked constraints, then the set of satisfied constraints represents a locally-predicate-better solution of the constraint hierarchy.*

DeltaBlue can be regarded as an algorithm that maintains the set of selected methods so that there are no blocked constraints.

We can explain the blocked constraint lemma from the viewpoint of GLP. A blocked constraint is an unsatisfied constraint such that there exists at least one weaker constraint that will be solved before the blocked one. Clearly, the absence of blocked constraints implies the sufficient condition (3.2) of Theorem 3.6 for GLP with local hierarchy comparators. Thus we can regard the blocked constraint lemma as a specialization of Theorem 3.6.

[5] The DeltaBlue algorithm cannot handle cyclic layouts of methods; it will halt if it finds a cycle.

Chapter 4

The *DETAIL* Constraint Solver

This chapter presents a constraint solver called *DETAIL*, which is the first local propagation algorithm adopting a global hierarchy comparator.

4.1 Overview

DETAIL is an incremental algorithm for solving constraint hierarchies based on local propagation. It always stores planning data instead of an appropriate ordered partition of the current hierarchy, and modifies the plan if a constraint is added to or removed from the hierarchy.

DETAIL handles multi-way equality constraints extended so that it can simultaneously satisfy or properly relax them, in addition to solving them individually as is with classical local propagation.

The need for simultaneous satisfaction of multiple constraints naturally arises in GUI applications. For example, consider a situation that a user attempts to move the midpoint m of two points a and b. The constraints on x-coordinates are as follows:

$$\mathsf{a.x} + 100 = \mathsf{b.x} \tag{4.1}$$

$$(\mathsf{a.x} + \mathsf{b.x})/2 = \mathsf{m.x} \tag{4.2}$$

$$\mathsf{m.x} = \mathsf{mouse.x}. \tag{4.3}$$

Obviously, constraint (4.3) should be solved as $\mathsf{m.x} \leftarrow \mathsf{mouse.x}$. Then, in traditional local propagation, constraint (4.2) should be either evaluated as $\mathsf{a.x} \leftarrow 2 * \mathsf{m.x} - \mathsf{b.x}$ or $\mathsf{b.x} \leftarrow 2 * \mathsf{m.x} - \mathsf{a.x}$. However, both of them results

in cyclic dependency because of (4.1), and therefore, local propagation fails. To resolve this problem, we need to simultaneously solve (4.1) and (4.2).

The demand for relaxing constraints also often occurs in applications with GUIs. A typical example is that a user moves an object a constrained to be on an almost horizontal or vertical line, e.g.

$$\mathsf{a.x} = 10 * \mathsf{a.y} \tag{4.4}$$

$$\mathsf{a.x} = \mathsf{mouse.x} \tag{4.5}$$

$$\mathsf{a.y} = \mathsf{mouse.y}. \tag{4.6}$$

If the user arbitrarily moves the mouse, all of these constraints cannot be satisfied at the same time. A solution may seem to discard either of constraint (4.5) or (4.6), which is possible with past constraint hierarchy solvers using locally-predicate-better by assigning weaker strengths to (4.5) and (4.6) than the strength of (4.4). However, if (4.5) is ignored, the user will find difficulty in moving the object a, because the mouse movement in its y-coordinate will make the movement of the x-coordinate of a 10-fold due to (4.4). Therefore, a natural solution is to relax constraints (4.5) and (4.6) using a global comparator such as least-squares-better.

To simultaneously satisfy or properly relax constraints, *DETAIL* maintains a set of *constraint cells* instead of an ordered partition into blocks. A constraint cell can be regarded as a block including output variables, where the constraints in the block are uniquely solved for the output variables. Also, it never shares variables with any other cells. *DETAIL* solves constraint cells with pluggable numerical modules called *subsolvers* using, e.g., Gaussian elimination.[1]

For example, to solve the constraint strong $x + y = 3$ for variable x, *DETAIL* yields a cell of strong $x + y = 3$ and x as shown in Figure 4.1 (a), where the circles and square represent the variables and constraint respectively, and the box with round corners indicate the cell. By contrast, to simultaneously solve strong $x + y = 3$ and weak $x - y = 1$, it generates a cell of the two constraints and the variables x and y as depicted in Figure 4.1 (b). Similarly, to relax strong $x = 0$ and strong $x = 2$, it produces a cell consisting of the two constraints and x as illustrated in Figure 4.1 (c).

By the definition of constraint cells, we can determine dependency among cells. Additionally, if we prohibit cyclic dependency, we can naturally identify the overall dependency among cells with a partial order among blocks. Then we can perform GLP in a 'unique' manner as is with conventional local

[1]The SkyBlue algorithm also realizes simultaneous satisfaction by invoking 'cycle solvers,' but provides no features for relaxing constraints [72, 73].

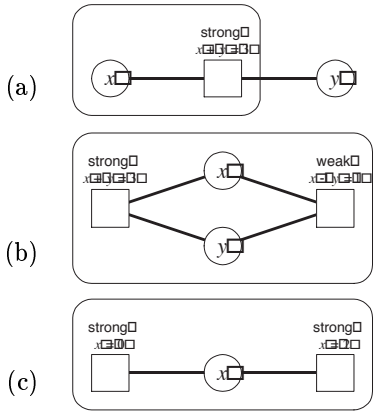

Figure 4.1: Constraint cells.

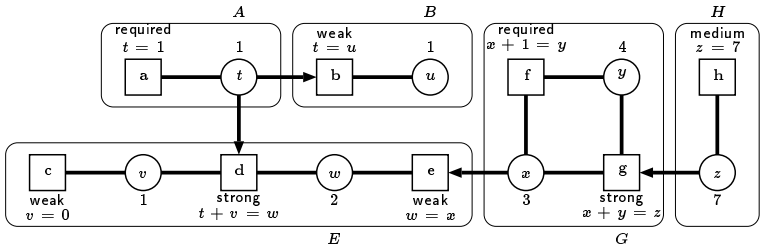

Figure 4.2: A configuration of constraint cells.

propagation. For example, consider the hierarchy with the constraints a, b, ..., h in Figure 4.2. Clearly, in the order respecting the cell dependencies, such as A, B, H, G, and E, we can uniquely solve constraints in each cell.

In the remainder of this chapter, we first present graph-formalized constraint hierarchies, and then describe the *DETAIL* algorithm and constraint solver.

4.2 Formulation

In this section, we formalize constraint hierarchies as graphs. The point is that we construct a certain kind of graphs that approximates satisfaction of constraint hierarchies by simulating GLP without algebraic concepts.

First, we define *constraint graphs* as follows:

Definition 4.1 (constraint graph). Given a set H of labeled constraints, the constraint graph of H is a bipartite graph $\langle X, H, E\rangle$ with sets X and H of nodes and a set E of edges, where

$$\begin{aligned} X &= \{x \in \boldsymbol{X} \mid \exists c/l \in H.\, x \in \boldsymbol{X}(c)\} \\ E &= \{\langle x, c/l\rangle \in X \times H \mid x \in \boldsymbol{X}(c)\}. \end{aligned}$$

In other words, a constraint graph consists of two kinds of nodes X and H corresponding to variables and labeled constraints respectively and edges E representing connections between variables and constraints.

In Definition 4.1, we consider that H is not a multi-set but an ordinary set, as opposed to the definition of constraint hierarchies in Chapter 2; that is, H does not contain multiple copies of a single labeled constraint. Similar to other local propagation algorithms, the *DETAIL* algorithm does not treat constraints as algebraic concepts. Therefore, it is not restrictive to suppose that labeled constraints are distinct objects.

Next, we define *constraint cells*, *cell configurations*, and *propagation graphs* as follows:

Definition 4.2 (constraint cell). Given a constraint graph $\langle X, H, E\rangle$, a constraint cell in $\langle X, H, E\rangle$ is a set of variables in X and labeled constraints in H.

Definition 4.3 (cell configuration). Given a constraint graph $\langle X, H, E\rangle$, a cell configuration for $\langle X, H, E\rangle$ is a set $\mathcal{P}$ of constraint cells in $\langle X, H, E\rangle$ such that

$$\forall \mathcal{B} \in \mathcal{P}.\, \forall \mathcal{B}' \in \mathcal{P}.\, \mathcal{B}' \neq \mathcal{B} \Rightarrow \mathcal{B} \cap \mathcal{B}' = \emptyset$$

and

$$\bigcup_{\mathcal{B} \in \mathcal{P}} \mathcal{B} = X \cup H.$$

Definition 4.4 (propagation graph). Given a constraint graph $\langle X, H, E\rangle$ and a cell configuration $\mathcal{P}$ for $\langle X, H, E\rangle$, a propagation graph of $\langle X, H, E\rangle$ with $\mathcal{P}$ is the quadruple $\langle X, H, E, \mathcal{P}\rangle$.

Intuitively, a cell configuration for a constraint graph consists of constraint cells containing variables and labeled constraints. Also, in a cell configuration, distinct cells do not share variables and labeled constraints, and all variables and constraints belong to some cells. We refer constraint graphs

with a cell configuration as propagation graphs. Clearly, we can illustrate propagation graphs as shown in Figure 4.2.

The following definitions provide conditions that "variables are constrained" and that "labeled constraints are active:"

Definition 4.5 (constrained variable). Given sets X, H, and E of variables, labeled constraints, and edges, $\mathit{constrained}(X, H, E)$ is defined as follows:

$$\mathit{constrained}(X, H, E) \equiv \forall X' \subseteq X. |X'| \leq |\Gamma_H(X', H, E)|$$

where

$$\Gamma_H(X', H, E) \equiv \{c/l \in H \mid \exists x \in X'. \langle x, c/l \rangle \in E\}.$$

Definition 4.6 (active labeled constraint). Given sets H, X, and E of labeled constraints, variables, and edges, $\mathit{active}(H, X, E)$ is defined as follows:

$$\mathit{active}(H, X, E) \equiv \forall H' \subseteq H. |H'| \leq |\Gamma_X(H', X, E)|$$

where

$$\Gamma_X(H', X, E) \equiv \{x \in X \mid \exists c/l \in H'. \langle x, c/l \rangle \in E\}.$$

Intuitively, $\mathit{constrained}(X, H, E)$ means that variables in X are constrained by labeled constraints in H, and $\mathit{active}(H, X, E)$ indicates that labeled constraints in H are active in constraining variables in X. Both of the definitions are based on an operation called 'perfect matching' in graph theory [13]. For a bipartite graph, the matching operation makes pairs of different kinds of nodes, and perfect matching is a matching providing all nodes of one kind with their partners. In Definition 4.5, if $\mathit{constrained}(X, H, E)$ holds, all variables in X can be matched with distinct labeled constraints in H; in other words, for each variable, some labeled constraint can output a value to it. Also, in Definition 4.6, if $\mathit{active}(H, X, E)$ holds, all constraints in H can be matched with different variables in X; that is, for each labeled constraint, there exists a variable that needs the constraint to obtain its value. The conditions for defining $\mathit{constrained}(X, H, E)$ and $\mathit{active}(H, X, E)$ are known as Hall's theorem in graph theory, which dictates existence of perfect matching.

Using $\mathit{active}(H, X, E)$ together with strengths, the following definitions classify labeled constraints into three situations, 'satisfied,' 'unsatisfied,' and 'relaxed:'

Definition 4.7 (satisfied labeled constraint). Given a constraint graph $\langle X, H, E\rangle$ and a constraint cell in $\langle X, H, E\rangle$, $H_{\mathrm{s}}(\mathcal{B}, \langle X, H, E\rangle)$ is defined as follows:

$$H_{\mathrm{s}}(\mathcal{B}, \langle X, H, E\rangle) \equiv \{c/l \in \mathcal{B} \mid \mathit{active}(H', \mathcal{B} \cap X, E)\}$$

where

$$H' \equiv \{c'/l' \in \mathcal{B} \cap H \mid l' \leq l\}.$$

Definition 4.8 (unsatisfied labeled constraint). Given a constraint graph $\langle X, H, E\rangle$ and a constraint cell in $\langle X, H, E\rangle$, $H_{\mathrm{u}}(\mathcal{B}, \langle X, H, E\rangle)$ is defined as follows:

$$H_{\mathrm{u}}(\mathcal{B}, \langle X, H, E\rangle) \equiv \{c/l \in \mathcal{B} \mid \neg\, \mathit{active}(H', \mathcal{B} \cap X, E)\}$$

where

$$H' \equiv \{c/l\} \cup \{c'/l' \in \mathcal{B} \cap H \mid l' < l\}.$$

Definition 4.9 (relaxed labeled constraint). Given a constraint graph $\langle X, H, E\rangle$ and a constraint cell in $\langle X, H, E\rangle$, $H_{\mathrm{r}}(\mathcal{B}, \langle X, H, E\rangle)$ is defined as follows:

$$H_{\mathrm{r}}(\mathcal{B}, \langle X, H, E\rangle) \equiv (\mathcal{B} \cap H) - (H_{\mathrm{s}}(\mathcal{B}, \langle X, H, E\rangle) \cup H_{\mathrm{u}}(\mathcal{B}, \langle X, H, E\rangle)).$$

Intuitively, we can make all satisfied labeled constraints active in their cell. By contrast, we cannot make any unsatisfied constraint active when we try to make all stronger constraints active. On the other hand, we can make a relaxed constraint active together with all stronger constraints.

The next definition presents an important class of propagation graphs called '*non-redundant propagation graph*:'

Definition 4.10 (non-redundant propagation graph). A propagation graph $\langle X, H, E, \mathcal{P}\rangle$ is non-redundant iff for any constraint cell $\mathcal{B}$ in $\mathcal{P}$, either of the following conditions hold:

$$|\mathcal{B} \cap X| = 1 \quad \wedge \quad |\mathcal{B} \cap H| = 0 \tag{4.7}$$

$$|\mathcal{B} \cap X| = 0 \quad \wedge \quad |\mathcal{B} \cap H| = 1 \tag{4.8}$$

$$\mathit{constrained}(\mathcal{B} \cap X, \mathcal{B} \cap H, E) \quad \wedge \quad H_{\mathrm{u}}(\mathcal{B}, \langle X, H, E\rangle) = \emptyset. \tag{4.9}$$

Non-redundant propagation graphs specially treat unconstrained (i.e. not constrained) variables and unsatisfied labeled constraints; each unconstrained variable constitutes a cell alone, and each unsatisfied labeled constraint belongs to a cell only for it. Therefore, we can easily find unconstrained variables and unsatisfied constraints in non-redundant propagation graphs.

The following definition gives a convenient notion that a cell is a successor to another cell in a propagation graph:

Definition 4.11 (successor constraint cell). Given a propagation graph $\langle X, H, E, \mathcal{P} \rangle$ and two constraint cells $\mathcal{B}$ and $\mathcal{B}'$ in $\mathcal{P}$, $succ(\mathcal{B}', \mathcal{B}, \langle X, H, E, \mathcal{P} \rangle)$ is defined as follows:

$$\mathcal{B}' \neq \mathcal{B} \wedge \exists x \in \mathcal{B}. \exists c/l \in \mathcal{B}'. \langle x, c/l \rangle \in E.$$

Intuitively, $succ(\mathcal{B}', \mathcal{B}, \langle X, H, E, \mathcal{P} \rangle)$ indicates that $\mathcal{B}'$ is a successor to $\mathcal{B}$ in $\langle X, H, E, \mathcal{P} \rangle$. The definition says that the value of the variable x in $\mathcal{B}$ is used by the labeled constraint c/l in $\mathcal{B}'$.

The next defines another important class of propagation graphs called '*acyclic proration graph*:'

Definition 4.12 (acyclic propagation graph). A propagation graph $\langle X, H, E, \mathcal{P} \rangle$ is acyclic iff there does not exist any sequence $\mathcal{B}_1$, $\mathcal{B}_2$, ..., $\mathcal{B}_n$ of constraint cells in $\mathcal{P}$ such that $\mathcal{B}_1 = \mathcal{B}_n$ and for any i s.t. $1 \leq i < n$,

$$succ(\mathcal{B}_{i+1}, \mathcal{B}_i, \langle X, H, E, \mathcal{P} \rangle).$$

Intuitively, acyclic propagation graphs do not have cyclic dependency of constraint cells; that is, cells are partially ordered.

Below we define *walkabout strengths* of constraint cells using *internal strengths*. As noted in Section 3.3, walkabout strengths were first introduced in the DeltaBlue algorithm [22, 58, 76], but we extend their definition to match them with our purpose:

Definition 4.13 (internal strength). Given a constraint cell $\mathcal{B}$, the internal strength of $\mathcal{B}$, denoted as $internal\_strength(\mathcal{B})$, is defined as follows:

$$internal\_strength(\mathcal{B}) \equiv \begin{cases} l_{\mathrm{w}} + 1 & \text{if } \mathcal{B} \cap (\boldsymbol{C} \times \boldsymbol{L}) = \emptyset \\ max\{l \in \boldsymbol{L} \mid \exists c \in \boldsymbol{C}. c/l \in \mathcal{B}\} & \text{otherwise.} \end{cases}$$

Definition 4.14 (walkabout strength). Given an acyclic propagation graph $\langle X, H, E, \mathcal{P}\rangle$ and a constraint cell $\mathcal{B}$ in $\langle X, H, E, \mathcal{P}\rangle$, the walkabout strength of $\mathcal{B}$, denoted as $walkabout\_strength(\mathcal{B}, \langle X, H, E, \mathcal{P}\rangle)$, is defined as follows:

$$\begin{aligned} & walkabout\_strength(\mathcal{B}, \langle X, H, E, \mathcal{P}\rangle) \\ \equiv\ & max(\{internal\_strength(\mathcal{B})\} \cup \\ & \quad \{l \in \boldsymbol{L} \mid \exists \mathcal{B}' \in \mathcal{P}.\, succ(\mathcal{B}, \mathcal{B}', \langle X, H, E, \mathcal{P}\rangle) \wedge \\ & \qquad walkabout\_strength(\mathcal{B}', \langle X, H, E, \mathcal{P}\rangle) = l\}). \end{aligned}$$

By definition, the internal strength of a constraint cell with one or more labeled constraints is the strength of the weakest constraint(s), and the internal strength of a cell with no labeled constraints is $l_{\mathrm{w}} + 1$, which is weaker than any strengths. Walkabout strengths of cells are defined so that they propagate the weakest internal strengths, that is, the strengths of the weakest constraints in some cells. Although walkabout strengths are defined recursively, they are always uniquely determined because they are defined for acyclic propagation graphs.

Now, we define *globally-graph-better (GGB) propagation graphs* so that such propagation graphs can be regarded to generate solutions with local propagation:

Definition 4.15 (GGB propagation graph). Given an acyclic, non-redundant propagation graph $\langle X, H, E, \mathcal{P}\rangle$, $\langle X, H, E, \mathcal{P}\rangle$ is a GGB propagation graph if

$$\begin{aligned} (4.10)\qquad & \forall \mathcal{B} \in \mathcal{P}.\, \forall c/l \in \mathcal{B}.\, c/l \notin H_{\mathrm{s}}(\mathcal{B}, \langle X, H, E\rangle) \\ & \Rightarrow\ (\forall \mathcal{B}' \in \mathcal{P}.\, succ(\mathcal{B}, \mathcal{B}', \langle X, H, E, \mathcal{P}\rangle) \\ & \qquad \Rightarrow walkabout\_strength(\mathcal{B}', \langle X, H, E, \mathcal{P}\rangle) < l). \end{aligned}$$

In other words, in GGB propagation graphs, any relaxed or unsatisfied constraints have only stronger constraints before them. Obviously, it corresponds to Theorem 3.4. Therefore, we claim that GGB propagation graphs simulate ordered partitions that obtain solutions.

As an example of GGB graphs, consider the propagation graph illustrated in Figure 4.3, where walkabout strengths are attached to cells. In this graph, although the weak constraints c and e in E are relaxed, the walkabout strengths required and medium of the preceding cells A and G indicate that all the forward constraints are stronger than weak. Therefore, we can obtain a correct solution by applying local propagation to the graph.

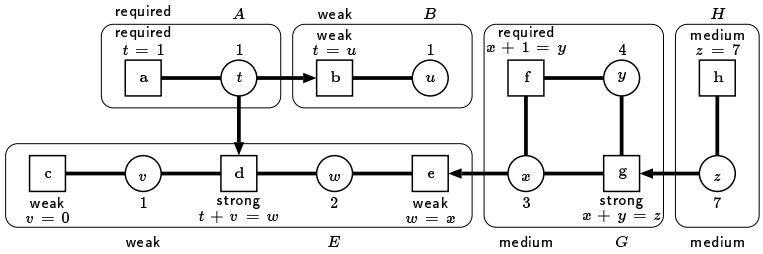

Figure 4.3: A GGB propagation graph.

4.3 Algorithm

This section describes the *DETAIL* algorithm for solving graph-formalized constraint hierarchies defined in the previous section.

In GUI applications, it is often necessary to repeatedly evaluate the same constraints in a propagation graph. For example, while a user is dragging a visual object, constraints related to the object need to be re-evaluated to cope with the user's real-time interaction. Therefore, most constraint solvers for GUIs, including *DETAIL*, adopt *two-phase constraint satisfaction*, which consists of *planning* and *execution*. Intuitively, the planning phase obtains an internal data structure for constraint systems, and the execution phase computes actual values for variables using the data structure constructed by planning.

4.3.1 The Planning Phase

The task of the planning phase of the *DETAIL* algorithm is to incrementally maintain a GGB propagation graph. Initially, the propagation graph has no variables and no constraints, and *DETAIL* appropriately modifies the graph when variables or constraints are added or removed.

Adding a Variable

First, we describe how to add a variable x to a GGB propagation graph $\langle X, H, E, \mathcal{P} \rangle$. It is only to add x to the set X of variables and also add a new cell with the variable, $\{x\}$, to the cell configuration $\mathcal{P}$. The resulting GGB propagation graph is

$$\langle X \cup \{x\}, H, E, \mathcal{P} \cup \{\{x\}\} \rangle.$$

Removing a Variable

Next, we provide how to remove an existing variable x from a GGB propagation graph $\langle X, H, E, \mathcal{P} \rangle$. *DETAIL* reports an error indicating that x cannot be removed if there exists some labeled constraint c/l such that $\langle x, c/l \rangle \in E$ (that is, x is referenced by a constraint). Otherwise, it generates the GGB graph

$$\langle X - \{x\}, H, E, \mathcal{P} - \{\{x\}\} \rangle.$$

Note that $\mathcal{P}$ must contain $\{x\}$ as an element since $\langle X, H, E, \mathcal{P} \rangle$ is GGB and therefore non-redundant.

Adding a Constraint

We show the algorithm for adding a new labeled constraint c/l to a GGB propagation graph $\langle X, H, E, \mathcal{P} \rangle$. There are the following three cases in this process:[2]

- c/l will remain unsatisfied since it is weak. In this case, the propagation graph will be almost unchanged, but will only be added a new cell with c/l.
- c/l will be relaxed since there are one or more conflicting constraints with the same strength l. In this case, c/l will be inserted to the cell with the equal-strength constraint(s) to be relaxed together. Also, the satisfied constraints between c/l and the equal-strength constraint(s) will be also added to the cell.
- c/l will be satisfied since it is strong enough to revoke another one or more weaker constraints. In this case, the dependencies between c/l and these weaker constraint(s) will be reversed to propagate the value from c/l, and the weaker constraint(s) will be relaxed or unsatisfied.

`add_constraint`$(c/l,\ \langle X, H, E, \mathcal{P} \rangle)$ in Figure 4.4 is the pseudo-code for adding a new labeled constraint c/l to a GGB propagation graph $\langle X, H, E, \mathcal{P} \rangle$. In this pseudo-code, $\mathcal{B}$.`walkabout_strength` indicates a field of $\mathcal{B}$ previously calculated by *DETAIL* as the walkabout strength of $\mathcal{B}$. Also, variables for cells such as $\mathcal{B}_{\text{new}}$ and $\mathcal{B}_{\text{vic}}$ are represented as pointers; even

[2] If we handle ordinary (not graph-formalized) constraint hierarchies, there will be the case that c/l is already satisfied. However, we do not consider such a case because of the formulation of constraint hierarchies for the *DETAIL* algorithm.

if their contents are modified, we mean that they are still pointing to the modified cells.

Now, we explain `add_constraint`. First, at lines 3 and 4, it modifies the sets H and E so that they reflect the information on the newly added constraint c/l. Next, at lines 5 and 6, it inserts a new constraint cell $\mathcal{B}_{\text{new}}$ only with c/l to make an initial configuration. At line 7, it obtains the weakest walkabout strengths l_{vic} of constraint cells preceding $\mathcal{B}_{\text{new}}$. At lines 8–20, it runs in three ways according to the relations between l_{vic} and l:

- Case $l_{\text{vic}} < l$ (lines 9 and 10). It leaves $\mathcal{B}_{\text{new}}$ as it is, and does not further modify the propagation graph.

- Case $l_{\text{vic}} = l$ (lines 13 and 14). It keeps $\mathcal{B}_{\text{new}}$ as $\mathcal{B}_{\text{vic}}$ to relax c/l later at line 22.

- Case $l_{\text{vic}} > l$ (lines 17–19). At line 17, it reverses dependency between $\mathcal{B}_{\text{new}}$ and some cell by calling `reverse_dependency`, which we describe later. Also, it sets the return value of `reverse_dependency` to $\mathcal{B}_{\text{vic}}$; if it is non-nil, it represents a cell that should be relaxed later at line 22. Then, at line 18, it merges cells that have cyclic dependency because of the previous reverse operation. It is a simple algorithm that only collects cyclic cells after $\mathcal{B}_{\text{new}}$ with marking and that merges them into a single cell. Then, at line 19, it traverses cells after $\mathcal{B}_{\text{new}}$ and recalculate their walkabout strengths.

Finally, at lines 21 and 22, if $\mathcal{B}_{\text{vic}}$ is non-nil, it traverses cells before $\mathcal{B}_{\text{vic}}$ and merges cells with walkabout strength l_{vic}.

Next, we briefly explain the algorithm of `reverse_dependency`, which is presented in Figure 4.5. Basically, it is a recursive function that traverses a single cell in each step. On entry into the function, $\mathcal{B}$ consists of only one constraint. First, at lines 3 to 6, it finds a preceding cell $\mathcal{B}_{\text{pred}}$ with walkabout strength l_{vic}, and moves a variable in $\mathcal{B}_{\text{pred}}$ to $\mathcal{B}$, which makes $\mathcal{B}$ a pair of a variable and a constraint. Next, it branches into the following three cases:

- Case $\mathcal{B}_{\text{pred}} = \emptyset$ (lines 8 and 9). It indicates that $\mathcal{B}_{\text{pred}}$ initially contains only one variable; that is, the variable is unconstrained. Therefore, the algorithm does not need to further modify the graph.

- Case $\mathit{internal\_strength}(\mathcal{B}_{\text{pred}}) = l_{\text{vic}}$ (line 12). It means that $\mathcal{B}_{\text{pred}}$ contains one or more constraints to be relaxed or unsatisfied. Accordingly, the algorithm separates the constraint(s) to be relaxed or unsatisfied

```
 1 add_constraint(c/l, ⟨X, H, E, P⟩)
 2 {
 3   H ← H ∪ {c/l};
 4   E ← E ∪ (X(c) × {c/l});
 5   B_new ← {c/l};
 6   P ← P ∪ {B_new};
 7   l_vic ← max{l' ∈ L | ∃B_pred ∈ P . succ(B_new, B_pred, ⟨X, H, E, P⟩)
         ∧ B_pred.walkabout_strength = l'};
 8   if l_vic < l then {
 9     B_new.walkabout_strength ← l;
10     B_vic ← nil;
11   }
12   else if l_vic = l then {
13     B_new.walkabout_strength ← l;
14     B_vic ← B_new;
15   }
16   else { // l_vic > l
17     B_vic ← reverse_dependency(B_new, ⟨X, H, E, P⟩, l_vic);
18     merge_cycles(B_new, ⟨X, H, E, P⟩);
19     update_walkabout_strengths(B_new, ⟨X, H, E, P⟩);
20   }
21   if B_vic ≠ nil then
22     merge_cells_to_relax(B_vic, l_vic);
23 }
```

Figure 4.4: Adding a constraint.

```
 1 reverse_dependency(B, ⟨X, H, E, P⟩, l_vic)
 2 {
 3   select B_pred ∈ P s.t. succ(B, B_pred, ⟨X, H, E, P⟩)
         ∧ B_pred.walkabout_strength = l_vic;
 4   select x ∈ B_pred s.t. ∃c/l ∈ B. ⟨x, c/l⟩ ∈ E;
 5   B_pred ← B_pred − {x};
 6   B ← B ∪ {x};
 7   if B_pred = ∅ then {
 8     P ← P − {B_pred};
 9     return nil;
10   }
11   else if internal_strength(B_pred) = l_vic then
12     return decompose_cell(B_pred, ⟨X, H, E, P⟩);
13   else {
14     select c/l ∈ B_pred s.t. ∃B_predpred ∈ P.
           succ(B_pred, B_predpred, ⟨X, H, E, P⟩)
           ∧ B_predpred.walkabout_strength = l_vic;
15     B_pred ← B_pred − {c/l};
16     decompose_cell(B_pred, ⟨X, H, E, P⟩);
17     B_new ← {c/l};
18     P ← P ∪ {B_new};
19     return reverse_dependency(B_new, ⟨X, H, E, P⟩, l_vic);
20   }
21 }
```

Figure 4.5: Reversing the dependency between constraint cells.

by invoking `decompose_cell`. Basically, `decompose_cell` finds a maximum matching of constraints with variables and generates a cell for each pair. To guarantee stronger constraints to be matched, it processes stronger constraints in advance. Since the propagation graph was initially GGB, it is possible to match all constraints stronger than l_{vic}. On exit, `decompose_cell` returns a cell consisting of a constraint with strength l_{vic}, and also `reverse_dependency` returns the cell to relax it later if possible.

- Otherwise (lines 14–20). First, at line 14, it finds a labeled constraint c/l in $\mathcal{B}_{\text{pred}}$ that bridges to a preceding cell with walkabout strength l_{vic}. Then, at lines 15 and 16, it deletes c/l from $\mathcal{B}_{\text{pred}}$ and decompose the resulting $\mathcal{B}_{\text{pred}}$ (in this case, each constraint in $\mathcal{B}_{\text{pred}}$ can always be matched with some variable). Next, at lines 17 and 18, it creates a new cell $\mathcal{B}_{\text{new}}$ with c/l, and finally, at line 19, it recursively calls itself to further process $\mathcal{B}_{\text{new}}$.

Now, we demonstrate the *DETAIL* algorithm by example. Figure 4.6 (a) illustrates the initial propagation graph, and suppose that we add a new constraint h, medium $z = 7$, to it. The current solution $z = 3$ conflicts with h, and the walkabout strength weak of G shows that there is one or more weak constraints in or before G.[3] Therefore, we must alter the propagation graph in the following steps:

1. First, move along the path from the new cell to the source of the walkabout strength, i.e. from H to E, reversing the dependency between them, as shown in Figure 4.6 (b). Now, the newly added constraint h becomes satisfied.

2. Next, merge cyclic dependencies generated from the previous step if any. In the example, we collapse the cycle of G' and F as illustrated in Figure 4.6 (c).

3. Third, check whether the 'victimized' cell E' has any forward cells with the same walkabout strength weak. Figure 4.6 (c) shows that D is such a cell. Since it violates the sufficient condition for GGB propagation graphs, merge all the transitively adjacent cells with the same walkabout strength, i.e. E', D, and C (but not B). Then, we obtain the final propagation graph in Figure 4.6 (d).

[3] As noted, the actual *DETAIL* algorithm does not check such a conflict of the added constraint with the current solution, but only examines the strength of the added constraint and walkabout strengths of related cells.

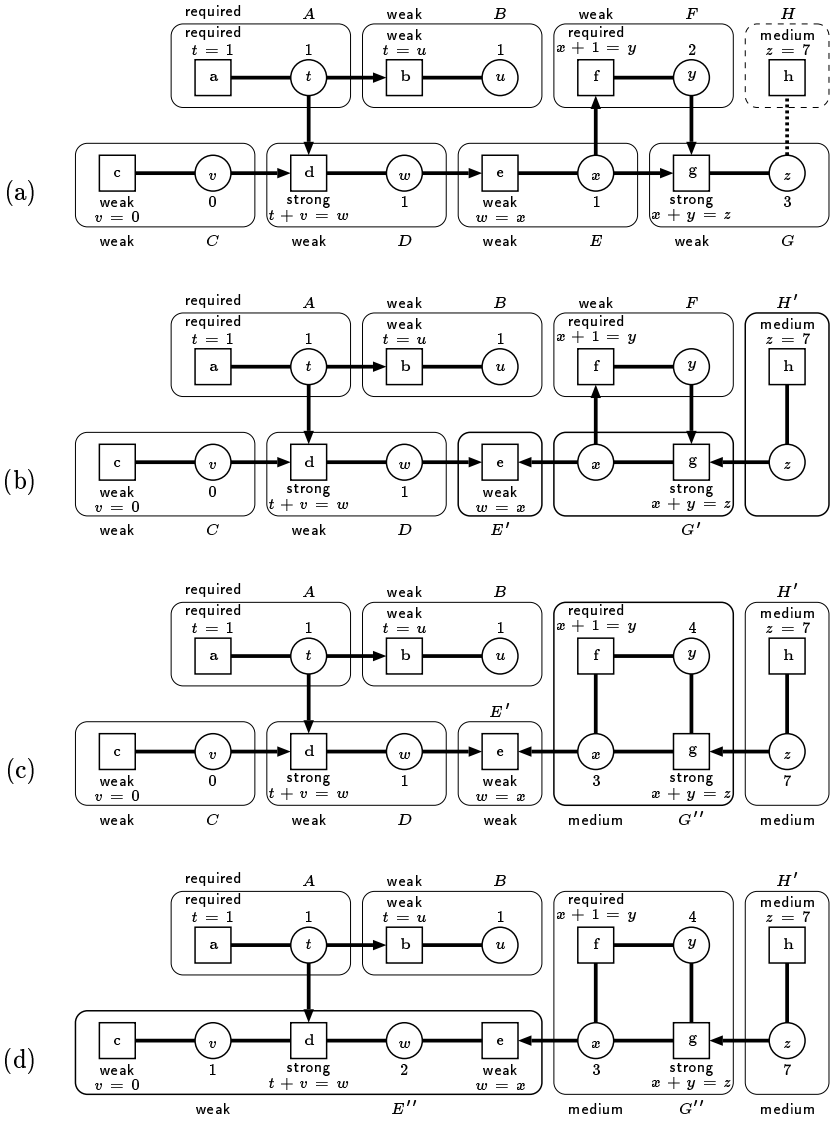

Figure 4.6: Adding a constraint to a constraint hierarchy.

Removing a Constraint

Finally, we describe how to remove an existing labeled constraint c/l from a GGB propagation graph $\langle X, H, E, \mathcal{P} \rangle$. We can consider the following three cases:

- c/l constitutes its own cell alone. It means that c/l is unsatisfied. Obviously, it is sufficient to only remove the cell from the graph.
- c/l belongs to a cell where each constraint is satisfied. It implies that removing c/l from the cell will generate an unconstrained variable. Therefore, it is necessary to find one or more relaxed constraints or one unsatisfied constraint that has been after c/l in the graph, and to try to satisfy or relax such constraints. The former process can be accomplished by traversing the propagation graph from the cell formerly with c/l and finding the strongest relaxed or unsatisfied constraints. The latter process can be realized by applying an algorithm similar to adding a constraint.
- c/l is contained by a cell that has one or more relaxed constraints. Then deleting c/l from the cell needs to produce a set of constraint cells that have satisfied or relaxed constraints. It can be done in a similar way to `decompose_cell`.

4.3.2 The Execution Phase

The task of the execution phase is to compute variable values based on the GGB propagation graph obtained by the planning phase. Its algorithm is to apply topological sort to the propagation graph and then to solve each constraint cell in the partial order obtained with the sort.

4.4 Implementation

Based on the algorithm in the previous section, we developed the *DETAIL* constraint solver in Objective C. It consists of two layers, a *main solver* and a set of *subsolvers*. The main solver maintains a GGB propagation graph in the planning phase and applies topological sort to it in the execution phase. By contrast, subsolvers obtain variable values by solving constraint systems locally in individual constraint cells. During the execution phase, the main solver invokes an appropriate subsolver for each cell, based on constraints in the cell.

In the implementation of the *DETAIL* constraint solver, we slightly modified the algorithm presented in the previous section so that it can handle local level comparators as well as global level comparators. As shown in our formulation of constraint hierarchies in Chapter 2, we can adopt a 'hybrid' hierarchy comparator with different level comparators to solve constraint hierarchies. This modification is theoretically realized by changing the sufficient condition of Definition 4.15 for local level comparators in a similar way to Theorem 3.6.

Using this technique, we implemented two subsolvers: one that treats linear equality constraints or multi-way constraints solved with the local level comparator with the predicate error function, and one that handles linear equality constraints satisfied with the least-squares level comparator.

DETAIL is actually used as a constraint solver in a programming-by-example system called IMAGE, which generates GUIs by generalizing multiple sets of application data and its visualization examples [61].

4.5 Performance Evaluation

Using the chain benchmark [76], we compared the performance of the *DETAIL* constraint solver implemented in Objective C with that of the DeltaBlue constraint solver implemented in C. We executed the benchmark program on a NeXTstation TurboColor workstation with a 33 MHz Motorola 68040 processor.

In the chain benchmark, the constraint hierarchy initially contains the required constraints $x_0 = x_1$, $x_1 = x_2$, ..., $x_{n-2} = x_{n-1}$ and the constraint weak $stay(x_0)$. The chain benchmark measures the planning time to add the constraint strong $edit(x_{n-1})$ to the hierarchy, and also measures the execution time to compute variable values when the value of x_{n-1} is changed through $edit(x_{n-1})$. Both of the planning phase and the execution phase are the worst cases where the overall propagation graph must be processed.

Table 4.1 shows the result: while the planning time of *DETAIL* is almost four times as long as that of DeltaBlue, the execution time is nearly twenty times as long. The main handicaps of *DETAIL* are the complex data structure of constraint cells, and dynamic binding of methods in Objective C.[4] We believe that dynamic binding caused slowdown in performance because the source program involves numerous message sends with dynamic binding. If we re-implement the *DETAIL* constraint solver in C++, its performance is expected to approach that of DeltaBlue.

[4] Objective C does not support static binding like C++.

n		1000	2000	3000	4000	5000
DeltaBlue	planning	67	166	250	350	434
	execution	2.5	4.3	6.7	8.7	10.8
DETAIL	planning	283	617	933	1183	1817
	execution	36.7	68.3	105.0	140.0	176.7

Table 4.1: Times in milliseconds to perform the chain benchmark.

Chapter 5

Hierarchical Linear Systems

This chapter formalizes hierarchical linear systems (HLSs) as a specialization of constraint hierarchies in linear equality constraints, and provides several basic algorithms for solving HLSs.

5.1 Motivation

Most constraint hierarchy solvers for GUIs, including *DETAIL*, are based on local propagation that handles multi-way constraints. By contrast, a small number of numerical solvers for GUIs are available. Therefore, most system developers currently use constraint solvers based on local propagation.

Although we showed that local propagation approach can be extended with the *DETAIL* algorithm, it has still disadvantages compared to numerical methods in handling algebraic constraints.[1] In fact, even for linear equations, local propagation solvers sometimes obtain incorrect solutions and in worse cases abort with fatal errors. These problems come from the essential difference between linear equations and multi-way constraints. Previously, the problem was mainly considered to be due to the inability of local propagation to simultaneously satisfy multiple constraints, which would appear as directed cycles of methods. Researches, including *DETAIL*, have been made to overcome this problem [72, 73], but they turned out to be insufficient to solve all the problems regarding linear equations.

[1] By local propagation, we mean algorithms for graph-formalized constraint systems, not generalized local propagation (GLP). Unlike local propagation algorithms, GLP can express various strategies for constraint satisfaction as a theoretical framework. For example, remember that GLP dictated the correctness of the refining method in Proposition 3.5

Even local propagation solvers that allows such cyclic dependencies, including *DETAIL* and SkyBlue [4], cannot handle cyclic dependencies with redundant or conflicting constraints. As an example, consider the following constraint hierarchy:

$$\text{strong} \quad x + y = 1 \tag{5.1}$$
$$\text{medium} \quad x + y = 3 \tag{5.2}$$
$$\text{weak} \quad x - y = 1. \tag{5.3}$$

Theoretically, satisfying (5.1) and (5.3), it yields a solution $x = 1$ and $y = 0$. However, local propagation solvers cannot find the correct solution because they select (5.1) and (5.2). In some cases, this problem is extremely serious. For instance, automatically generated constraints tend to be redundant or inconsistent, which we actually experienced in development of the programming-by-example system IMAGE [61]. However, redundancy and inconsistency are essentially difficult to deal with in local propagation algorithms.

Local propagation also embodies a problem related to the efficiency of incremental satisfaction of simultaneous constraints. Local propagation solvers supporting simultaneous constraints, e.g. SkyBlue and *DETAIL*, generate large cycles or constraint cells when they need to simultaneously satisfy many constraints. The problem is that they cannot always create, modify, or destroy such large cycles or cells incrementally: adding a new constraint to a system with no cycles may suddenly yield a large cycle; also, removing an exiting constraint from a system with a large cycle may decompose the cycle into pieces of constraints. Nevertheless, whenever a solver encounters such a case, it needs to perform special treatment of a cycle, which is considered to degenerate its performance.

In addition to the demand for addressing such problems, we are now encountering a new trend that constraint hierarchy algorithms are differentiated into two levels: 'meta' algorithms [10] and specialized algorithms [5]. A meta algorithm maintains the whole constraint hierarchy. Intuitively, it divides the hierarchy into a set of 'sub-hierarchies' based on its graph topology, making appropriate specialized algorithms actually solve the sub-hierarchies. Since developing an efficient, general algorithm is quite difficult, this trend will likely be the mainstream, strengthening the need for efficient specialized algorithms.

In this chapter, we show how to solve hierarchies of linear equality constraints efficiently. For this purpose, we construct the theory of *hierarchical linear systems* (HLSs), which can be viewed as a 'sub-theory' of constraint

hierarchies specialized in linear equality constraints. To our knowledge, this theory is the first sub-theory for linear equality constraints, although there have been ones proposed for multi-way constraints [72]. HLSs have a formally strict relation with the original constraint hierarchies. After presenting the theory, we give basic algorithms for HLSs.

5.2 Totally-Ordered Hierarchical Constraint Systems

Before presenting HLSs, we formalize *totally-ordered hierarchical constraint systems* (TOHCSs). In their definition, we do not focus on specific kinds of constraints, which is similar to the formulation of constraint hierarchies. In the next section, we formulate HLSs by specializing TOHCSs in linear equality constraints.

The definition of TOHCSs is the following:

Definition 5.1 (TOHCS). A TOHCS is a pair $\langle C_0, C\rangle$ of a set $C_0 = \{c_1^0, c_2^0, \ldots, c_{m_0}^0\}$ of m_0 constraints and an ordered set $C = \{c_1, c_2, \ldots, c_m\}$ of m constraints.

Intuitively, c_1^0, c_2^0, ..., $c_{m_0}^0$ are required constraints[2], and c_1, c_2, ..., c_m are preferential ones, where the first constraint has the strongest effect on determining solutions of the TOHCS, and the more backward the constraint, the weaker its effect; we strictly define it in Definition 5.2.

In the same way as constraint hierarchies, we evaluate errors of constraints using an error function e. Given a constraint c and variable assignment θ, $e(c,\theta)$ indicates the error of c for variable values presented with θ.

Using e, we define solutions of TOHCSs as follows:

Definition 5.2 (solution). Given an TOHCS $\langle C_0, C\rangle$, the solution set of $\langle C_0, C\rangle$ is

$$S(\langle C_0, C\rangle) \equiv \{\theta \in S_0 \mid \forall \theta' \in S_0.\, E(C,\theta) \leq_{\mathrm{lex}} E(C,\theta')\}$$

where

$$\begin{aligned} S_0 &= \{\theta \in \boldsymbol{\Theta} \mid \forall c_i^0 \in C_0.\, e(c_i^0,\theta) = 0\} \\ E(C,\theta'') &\equiv (e(c_1,\theta''), e(c_2,\theta''), \ldots, e(c_m,\theta'')) \end{aligned}$$

[2] Although we omitted required constraints in our formulation of constraint hierarchies for simplicity, we incorporate required ones in TOHCSs and HLSs; in solving HLSs, required constraints are useful for improving the efficiency.

and $\leq_{\text{lex}}$ is a lexicographic order, i.e. $\boldsymbol{u} \leq_{\text{lex}} \boldsymbol{v}$ is

$$\boldsymbol{u} =_{\text{lex}} \boldsymbol{v} \equiv \forall i.\, |u_i| = |v_i|$$

or

$$\boldsymbol{u} <_{\text{lex}} \boldsymbol{v} \equiv \exists i.\, (\forall i' < i.\, |u_{i'}| = |v_{i'}|) \wedge |u_i| < |v_i|.$$

Intuitively, $\leq_{\text{lex}}$ 'hierarchically' compares two vectors expressing errors, and the solution set of $\langle C_0, C \rangle$ is the set of all assignments satisfying C_0 that result in the minimum errors with respect to C in the sense of $\leq_{\text{lex}}$.

TOHCSs are simple HCSs that hold preferential constraints in total order. Unlike constraint hierarchies, they have no levels that contain constraints with equal preferences except the required one. However, if we solve constraint hierarchies with locally-better, we can convert them into TOHCSs as shown in the next proposition:

Proposition 5.3. *Let H be a constraint hierarchy whose level l ($0 \leq l \leq l_{\text{w}}$) consists of m_l constraints c_i^l ($1 \leq i \leq m_l$), and S be the set of all locally-better solutions of H. Let $\langle C_0, C \rangle$ be a TOHCS, where*

$$C_0 = \{c_1^0, c_2^0, \ldots, c_{m_0}^0\}$$

the i-th constraint in C is $c^l_{i-(m_1+\cdots+m_{l-1})}$ for l s.t.

$$m_1 + \cdots + m_{l-1} < i \leq m_1 + \cdots + m_l$$

and S' be the solution set of $\langle C_0, C \rangle$. Then $S' \subseteq S$.

Proof. Let $\theta \in S'$. Assume, for contradiction, $\theta \notin S$. Then there exists some θ' such that $\theta' \overset{H}{<} \theta$. Hence, for some l, $\forall l' < l.\, \theta' \overset{H/l'}{\sim} \theta$ and $\theta' \overset{H/l}{<} \theta$ hold, i.e.

$$\begin{aligned}(\forall l' < l.\, \forall i.\, e(c_i^{l'}, \theta') = e(c_i^{l'}, \theta)) \quad &\wedge \\ (\forall i.\, e(c_i^l, \theta') \leq e(c_i^l, \theta)) \quad &\wedge \quad (\exists i.\, e(c_i^l, \theta') < e(c_i^l, \theta)).\end{aligned}$$

Let i^* be the minimum index that satisfies

$$e(c_{i^*}^l, \theta') < e(c_{i^*}^l, \theta).$$

Then, for all $i < i^*$,

$$e(c_i^l, \theta') = e(c_i^l, \theta).$$

Thus $E(C, \theta') <_{\text{lex}} E(C, \theta)$, which is contradiction to $\theta \in S'$. Therefore, $\theta \in S$. □

In other words, the TOHCS obtained by 'serializing' the preferential constraints in a constraint hierarchy yields a subset of solutions of the original hierarchy. Therefore, if we do not need all solutions, we can use TOHCSs instead of constraint hierarchies. Such a situation is common in various applications, e.g. graphical user interfaces that usually need only one solution. Therefore, we believe that the notion of TOHCSs is useful as an alternative method to handle constraint hierarchies.

Essentially, TOHCSs are equivalent to ordered constraint hierarchies [94], which make constraints totally ordered inside levels. However, we believe that eliminating the concept of levels is more convenient for designing algorithms, which we show in the rest of this chapter.

5.3 The Theory of Hierarchical Linear Systems

In this section, we specialize TOHCSs in linear equality constraints, and derive their unique properties from the fact that they consist of linear equations.

5.3.1 Formulation

First, we define *hierarchical linear systems* (HLSs):

Definition 5.4 (HLS). Let $\boldsymbol{x}$ be a column vector of variables. An HLS is an TOHCS $\langle C_0, C\rangle$, where C_0 is a set of linear equality constraints $\boldsymbol{a}_i^0\boldsymbol{x} = c_i^0$, and C is an ordered set of linear equality constraints $\boldsymbol{a}_i\boldsymbol{x} = c_i$.

Since HLSs are TOHCSs, Definition 5.2 determines their solutions. For this purpose, we need error functions for linear equations. In the same way as constraint hierarchies, we can adopt the predicate and metric error functions. Given linear equality constraint $\boldsymbol{a}\boldsymbol{x} = c$, the metric error function e is defined as follows:

$$e(\boldsymbol{a}\boldsymbol{x} = c, \theta) \equiv |\boldsymbol{a}(\theta x_1\ \theta x_2\ \ldots\ \theta x_n)^{\mathrm{T}} - c|$$

Also, the predicate error function e is given as follows:

$$e(\boldsymbol{a}\boldsymbol{x} = c, \theta) \equiv \begin{cases} 0 & \text{if } \boldsymbol{a}(\theta x_1\ \theta x_2\ \ldots\ \theta x_n)^{\mathrm{T}} = c \\ 1 & \text{otherwise} \end{cases}$$

Since writing $(\theta x_1\ \theta x_2\ \ldots\ \theta x_n)^{\mathrm{T}}$ is rather complicated, in the rest of this dissertation, we also write $\boldsymbol{x}$ to indicate a vector with actual values, without using an assignment.

For convenience, we use an abbreviated notation for HLSs. Given an HLS $\langle C_0, C\rangle$, we express C_0 as $(A_0\ \boldsymbol{c}_0)$, i.e.

$$(A_0\ \boldsymbol{c}_0) = \begin{pmatrix} \boldsymbol{a}^0_1 & c^0_1 \\ \boldsymbol{a}^0_2 & c^0_2 \\ \vdots & \vdots \\ \boldsymbol{a}^0_{m_0} & c^0_{m_0} \end{pmatrix} = \left(\begin{array}{cccc|c} a^0_{11} & a^0_{12} & \cdots & a^0_{1n} & c^0_1 \\ a^0_{21} & a^0_{22} & \cdots & a^0_{2n} & c^0_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a^0_{m_0 1} & a^0_{m_0 2} & \cdots & a^0_{m_0 n} & c^0_{m_0} \end{array}\right)$$

and represent C as $(A\ \boldsymbol{c})$, i.e.

$$(A\ \boldsymbol{c}) = \begin{pmatrix} \boldsymbol{a}_1 & c_1 \\ \boldsymbol{a}_2 & c_2 \\ \vdots & \vdots \\ \boldsymbol{a}_m & c_m \end{pmatrix} = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & c_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & c_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & c_m \end{array}\right).$$

Thus we denote such an HLS as $\langle (A_0\ \boldsymbol{c}_0), (A\ \boldsymbol{c})\rangle$. We refer $\boldsymbol{a} = (a_1\ a_2\ \cdots\ a_i)$ as the *coefficient vector* of $\boldsymbol{a}\boldsymbol{x} = c$, A_0 and A as *coefficient matrices*, and $(A_0\ \boldsymbol{c}_0)$ and $(A\ \boldsymbol{c})$ as *extended coefficient matrices*.

For simplicity, we often focus on HLSs consisting only of preferential constraints. In such a case, we simply write $(A\ \boldsymbol{c})$ as an HLS.

5.3.2 Properties of Hierarchical Linear Systems

In this subsection, we derive unique properties of HLSs from the fact that they consist of linear equations. For simplicity, we consider HLSs consisting only of preferential constraints.

First, we show a necessary and sufficient condition for obtaining solutions:

Theorem 5.5. *For any HLS $(A\ \boldsymbol{c})$, $\boldsymbol{x}$ is a solution of $(A\ \boldsymbol{c})$ iff*

$$\forall i.\ \neg\, hdep(A, i) \Rightarrow \boldsymbol{a}_i\boldsymbol{x} = c_i \tag{5.4}$$

where $hdep(A, i)$ (the i-th row of A is hierarchically dependent) is defined as follows:

$$hdep(A, i) \equiv \exists\alpha_1 \exists\alpha_2 \cdots \exists\alpha_{i-1}.\, \boldsymbol{a}_i = \alpha_1\boldsymbol{a}_1 + \alpha_2\boldsymbol{a}_2 + \cdots + \alpha_{i-1}\boldsymbol{a}_{i-1}.$$

Proof. First, we show that $\boldsymbol{x}$ is a solution if (5.4) holds. Let $\boldsymbol{x}$ satisfy (5.4). Assume, for contradiction, that $\boldsymbol{x}$ is not a solution. Then there exists some $\boldsymbol{y}$ such that

$$A\boldsymbol{y} - \boldsymbol{c} <_{\text{lex}} A\boldsymbol{x} - \boldsymbol{c}$$

i.e., for some i,

$$\forall i' < i. |\boldsymbol{a}_{i'}\boldsymbol{y} - c_{i'}| = |\boldsymbol{a}_{i'}\boldsymbol{x} - c_{i'}| \wedge |\boldsymbol{a}_i\boldsymbol{y} - c_i| < |\boldsymbol{a}_i\boldsymbol{x} - c_i|.$$

Since $\boldsymbol{a}_i\boldsymbol{x} \neq c_i$, the i-th row must be hierarchically dependent. Selecting $i_1, i_2, \ldots, i_l < i$ such that the i_k-th row is hierarchically independent, we can certainly satisfy

$$\boldsymbol{a}_i = \alpha_{i_1}\boldsymbol{a}_{i_1} + \alpha_{i_2}\boldsymbol{a}_{i_2} + \cdots + \alpha_{i_l}\boldsymbol{a}_{i_l}.$$

As $\boldsymbol{a}_{i_k}\boldsymbol{x} = c_{i_k}$ by (5.4),

$$|\boldsymbol{a}_{i_k}\boldsymbol{y} - c_{i_k}| = |\boldsymbol{a}_{i_k}\boldsymbol{x} - c_{i_k}| = 0$$

which follows $\boldsymbol{a}_{i_k}\boldsymbol{y} = c_{i_k}$. Hence

$$\begin{aligned} \boldsymbol{a}_i\boldsymbol{y} &= (\alpha_{i_1}\boldsymbol{a}_{i_1} + \cdots + \alpha_{i_l}\boldsymbol{a}_{i_l})\boldsymbol{y} \\ &= \alpha_{i_1}c_{i_1} + \cdots + \alpha_{i_l}c_{i_l} \\ &= (\alpha_{i_1}\boldsymbol{a}_{i_1} + \cdots + \alpha_{i_l}\boldsymbol{a}_{i_l})\boldsymbol{x} \\ &= \boldsymbol{a}_i\boldsymbol{x} \end{aligned}$$

which is contradiction to $|\boldsymbol{a}_i\boldsymbol{y} - c_i| < |\boldsymbol{a}_i\boldsymbol{x} - c_i|$. Therefore $\boldsymbol{x}$ is a solution.

Next, we prove that (5.4) holds if $\boldsymbol{x}$ is a solution. Let $\boldsymbol{x}$ be a solution. Assume, for contradiction, $\boldsymbol{x}$ does not satisfy (5.4). Let i^* be the minimum index such that

$$\boldsymbol{a}_{i^*}\boldsymbol{x} \neq c_{i^*}$$

although the i^*-th row is hierarchically independent. Choose all $i_1, i_2, \ldots, i_l < i^*$ such that the i_k-th row is hierarchically independent, which follows $\boldsymbol{a}_{i_k}\boldsymbol{x} = c_{i_k}$. Then there exists at least one $\boldsymbol{y}$ such that $\boldsymbol{a}_{i_1}\boldsymbol{y} = c_{i_1}$, $\ldots$, $\boldsymbol{a}_{i_l}\boldsymbol{y} = c_{i_l}$, $\boldsymbol{a}_{i^*}\boldsymbol{y} = c_{i^*}$ because $\boldsymbol{a}_{i_1}, \cdots, \boldsymbol{a}_{i_l}, \boldsymbol{a}_{i^*}$ are linearly independent. Now, for all $i < i^*$, we can consider either of the following two cases:

1. If the i-th row is hierarchically independent,

$$|\boldsymbol{a}_i\boldsymbol{y} - c_i| = |\boldsymbol{a}_i\boldsymbol{x} - c_i| = 0.$$

2. Otherwise, for some $i_{l'} < i$, we can satisfy

$$\boldsymbol{a}_i = \alpha_{i_1}\boldsymbol{a}_{i_1} + \cdots + \alpha_{i_{l'}}\boldsymbol{a}_{i_{l'}}.$$

Hence,

$$\begin{aligned} \boldsymbol{a}_i\boldsymbol{y} &= (\alpha_{i_1}\boldsymbol{a}_{i_1} + \cdots + \alpha_{i_{l'}}\boldsymbol{a}_{i_{l'}})\boldsymbol{y} \\ &= c_{i_1} + \cdots + c_{i_{l'}} \\ &= (\alpha_{i_1}\boldsymbol{a}_{i_1} + \cdots + \alpha_{i_{l'}}\boldsymbol{a}_{i_{l'}})\boldsymbol{x} \\ &= \boldsymbol{a}_i\boldsymbol{x} \end{aligned}$$

which implies

$$|\boldsymbol{a}_i\boldsymbol{y} - c_i| = |\boldsymbol{a}_i\boldsymbol{x} - c_i|.$$

Also, as $\boldsymbol{a}_{i^*}\boldsymbol{y} = c_{i^*}$,

$$|\boldsymbol{a}_{i^*}\boldsymbol{y} - c_{i^*}| < |\boldsymbol{a}_{i^*}\boldsymbol{x} - c_{i^*}|.$$

Thus $|A\boldsymbol{y} - \boldsymbol{c}| <_{\text{lex}} |A\boldsymbol{x} - \boldsymbol{c}|$, which is contradiction to that $\boldsymbol{x}$ is a solution. Therefore (5.4) holds. □

Intuitively, solutions satisfy each hierarchically independent row, whose coefficient vector is linearly independent of those of all the upper or stronger ones. Conversely, if an equation has a dependent coefficient vector, it exhibits either inconsistency that we must discard, or redundancy that we may ignore.[3]

Unlike Definition 5.2, Theorem 5.5 provides a direct way to obtain solutions; that is, it describes which rows to actually solve. For example, consider an HLS with $x + y + z = 3$, $x = 0$, $x + y + z = 4$, $y + z = 3$, and $y = 1$ in this order, whose extended coefficient matrix is

$$\left(\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 4 \\ 0 & 1 & 1 & 3 \\ 0 & 1 & 0 & 1 \end{array}\right). \tag{5.5}$$

By Theorem 5.5, any solution must satisfy the first equation $x + y + z = 3$ since its coefficient vector $(1\ 1\ 1)$ is hierarchically independent.[4] Also,

[3] By Theorem 5.5, we do not need to minimize errors of hierarchically dependent equations; we only need to satisfy all hierarchically independent equations.

[4] Definition 5.2 suggests that the first row of a coefficient matrix is hierarchically independent if its coefficient vector is not $(0\ 0\ \cdots\ 0)$. Otherwise, the first equation is either always satisfiable, e.g., $0x + 0y = 0$, or never satisfiable, e.g., $0x + 0y = 1$. Therefore, we do not need to consider the case.

the second equation $x = 0$ must be satisfied since $(1\ 0\ 0)$ is independent. However, we do not satisfy the third one $x + y + z = 4$ because $(1\ 1\ 1)$ $(= (1\ 1\ 1) + 0 \cdot (1\ 0\ 0))$ is dependent. In this case, it is inconsistent with the first one. Also, the fourth $y + z = 3$ is dependent (as $(0\ 1\ 1) = (1\ 1\ 1) - 1 \cdot (1\ 0\ 0) + 0 \cdot (1\ 1\ 1)$). In this case, the equation is redundant because it can be composed of the first and second ones. Finally, the fifth equation is independent and therefore satisfied. We thus obtain the unique solution $x = 0$, $y = 1$, and $z = 2$, that is, the solution set is $\{(0\ 1\ 2)^{\mathrm{T}}\}$.

Next, we present a simple but useful lemma that we call *elevation*, which allows us to transform an extended coefficient matrix by 'elevating' a hierarchically independent row to an upper position:

Lemma 5.6 (elevation). *Let $(A\ \boldsymbol{c})$ be an arbitrary HLS, i an arbitrary index such that $\neg\, hdep(A, i)$, and i' an arbitrary index such that $i' < i$. Let $(B\ \boldsymbol{d})$ be an $m \times n$ matrix such that the first to $(i'-1)$-th rows of $(B\ \boldsymbol{d})$ are the first to $(i'-1)$-th rows of $(A\ \boldsymbol{c})$, the i'-th row of $(B\ \boldsymbol{d})$ is the i-th row of $(A\ \boldsymbol{c})$, the $(i'+1)$ to i-th rows of $(B\ \boldsymbol{d})$ are the i'-th to $(i-1)$-th rows of $(A\ \boldsymbol{c})$, and the $(i+1)$ to m-th rows of $(B\ \boldsymbol{d})$ are the $(i+1)$-th to m-th rows of $(A\ \boldsymbol{c})$, i.e.,*

$$(B\ \boldsymbol{d}) = \begin{pmatrix} \vdots & \vdots \\ \boldsymbol{b}_{i'-1} & d_{i'-1} \\ \boldsymbol{b}_{i'} & d_{i'} \\ \boldsymbol{b}_{i'+1} & d_{i'+1} \\ \vdots & \vdots \\ \boldsymbol{b}_{i} & d_{i} \\ \boldsymbol{b}_{i+1} & d_{i+1} \\ \vdots & \vdots \end{pmatrix} = \begin{pmatrix} \vdots & \vdots \\ \boldsymbol{a}_{i'-1} & c_{i'-1} \\ \boldsymbol{a}_{i} & c_{i} \\ \boldsymbol{a}_{i'} & c_{i'} \\ \vdots & \vdots \\ \boldsymbol{a}_{i-1} & c_{i-1} \\ \boldsymbol{a}_{i+1} & c_{i+1} \\ \vdots & \vdots \end{pmatrix}.$$

Then the solution set of $(B\ \boldsymbol{d})$ is equal to that of $(A\ \boldsymbol{c})$.

Proof. Let S and T be the solution sets of $(A\ \boldsymbol{c})$ and $(B\ \boldsymbol{d})$ respectively. First, we show $S \subseteq T$. Let $\boldsymbol{x} \in S$. To prove that $\neg\, hdep(B, i'')$ implies $\boldsymbol{b}_{i''}\boldsymbol{x} = d_{i''}$, we consider the following four cases:

1. *Case $i'' < i'$.* Assume $\neg\, hdep(B, i'')$. Clearly, $\neg\, hdep(A, i'')$. Since $\boldsymbol{a}_{i''}\boldsymbol{x} = c_{i''}$, $\boldsymbol{b}_{i''}\boldsymbol{x} = d_{i''}$.

2. *Case $i'' = i'$.* Since $\neg\, hdep(A, i)$, $\boldsymbol{a}_i\boldsymbol{x} = c_i$. Therefore $\boldsymbol{b}_{i'}\boldsymbol{x} = d_{i'}$.

3. *Case $i' < i'' \leq i$.* We prove the contrapositive: Assume $\boldsymbol{b}_{i''}\boldsymbol{x} \neq d_{i''}$. Then $\boldsymbol{a}_{i''-1}\boldsymbol{x} \neq c_{i''-1}$. As $hdep(A, i''-1)$, there exist some

$\alpha_1, \ldots, \alpha_{i''-2}$ such that

$$\boldsymbol{a}_{i''-1} = \alpha_1 \boldsymbol{a}_1 + \cdots + \alpha_{i''-2} \boldsymbol{a}_{i''-2}.$$

Hence

$$\begin{aligned} \boldsymbol{b}_{i''} &= \alpha_1 \boldsymbol{b}_1 + \cdots + \boldsymbol{a}_{i'-1} \boldsymbol{b}_{i'-1} + 0 \cdot \boldsymbol{b}_{i'} \\ &\quad + \alpha_{i'} \boldsymbol{b}_{i'+1} + \cdots + \alpha_{i''-2} \boldsymbol{b}_{i''-1}. \end{aligned}$$

Thus $hdep(B, i'')$.

4. *Case* $i'' > i$. We prove the contrapositive: Assume $\boldsymbol{b}_{i''}\boldsymbol{x} \neq d_{i''}$. Then $\boldsymbol{a}_{i''}\boldsymbol{x} \neq c_{i''}$. As $hdep(A, i'')$, there exist some $\alpha_1, \ldots, \alpha_{i''-1}$ such that

$$\boldsymbol{a}_{i''} = \alpha_1 \boldsymbol{a}_1 + \cdots + \alpha_{i''-1} \boldsymbol{a}_{i''-1}.$$

Hence

$$\begin{aligned} \boldsymbol{b}_{i''} &= \alpha_1 \boldsymbol{b}_1 + \cdots + \alpha_{i'-1} \boldsymbol{b}_{i'-1} + \alpha_i \boldsymbol{b}_{i'} \\ &\quad + \alpha_{i'} \boldsymbol{b}_{i'+1} + \cdots + \alpha_{i-1} \boldsymbol{b}_i \\ &\quad + \alpha_{i+1} \boldsymbol{b}_{i+1} + \cdots + \alpha_{i''-1} \boldsymbol{b}_{i''-1}. \end{aligned}$$

Accordingly $hdep(B, i'')$.

In all the cases, $\neg\, hdep(B, i'')$ implies $\boldsymbol{b}_{i''}\boldsymbol{x} = d_{i''}$. Therefore $\boldsymbol{x} \in T$.

Next, we give $S \supseteq T$. Let $\boldsymbol{x} \in T$. To prove that $\neg\, hdep(A, i'')$ implies $\boldsymbol{a}_{i''}\boldsymbol{x} = c_{i''}$, we suppose the following four cases:

1. *Case* $i'' < i'$. Assume $\neg\, hdep(A, i'')$. Clearly, $\neg\, hdep(B, i'')$. Since $\boldsymbol{b}_{i''}\boldsymbol{x} = d_{i''}$, $\boldsymbol{a}_{i''}\boldsymbol{x} = c_{i''}$.

2. *Case* $i' \leq i'' < i$. We prove the contrapositive: Assume $\boldsymbol{a}_{i''}\boldsymbol{x} \neq c_{i''}$. Then $\boldsymbol{b}_{i''+1}\boldsymbol{x} \neq d_{i''+1}$. As $hdep(B, i''+1)$, there exist some $\beta_1, \ldots, \beta_{i''}$ such that

$$\boldsymbol{b}_{i''+1} = \beta_1 \boldsymbol{b}_1 + \cdots + \beta_{i''} \boldsymbol{b}_{i''}.$$

Hence

$$\begin{aligned} \boldsymbol{a}_{i''} &= \beta_1 \boldsymbol{a}_1 + \cdots + \beta_{i'-1} \boldsymbol{a}_{i'-1} \\ &\quad + \beta_{i'+1} \boldsymbol{a}_{i'} + \cdots + \beta_{i''} \boldsymbol{a}_{i''-1} + \beta_{i'} \boldsymbol{a}_i. \end{aligned}$$

Since $\neg\, hdep(A, i)$, $\beta_{i'} = 0$ is required. Thus $hdep(A, i'')$.

3. *Case* $i'' = i$. Since $\neg\, hdep(A, i)$, $\neg\, hdep(B, i')$. Hence $\boldsymbol{b}_{i'}\boldsymbol{x} = d_{i'}$. Therefore $\boldsymbol{a}_i\boldsymbol{x} = c_i$.

4. *Case* $i'' > i$. We prove the contrapositive: Assume $\boldsymbol{a}_{i''}\boldsymbol{x} \neq c_{i''}$. Then $\boldsymbol{b}_{i''}\boldsymbol{x} \neq d_{i''}$. As $hdep(B, i'')$, there exist some $\beta_1, \ldots, \beta_{i''-1}$ such that

$$\boldsymbol{b}_{i''} = \beta_1\boldsymbol{b}_1 + \cdots + \beta_{i''-1}\boldsymbol{b}_{i''-1}.$$

Hence

$$\begin{aligned}\boldsymbol{a}_{i''} \;=\; & \beta_1\boldsymbol{a}_1 + \cdots + \beta_{i'-1}\boldsymbol{a}_{i'-1} \\ & + \beta_{i'+1}\boldsymbol{a}_{i'} + \cdots + \beta_i\boldsymbol{a}_{i-1} + \beta_{i'}\boldsymbol{a}_i \\ & + \beta_{i+1}\boldsymbol{a}_{i+1} + \cdots + \beta_{i''-1}\boldsymbol{a}_{i''-1}.\end{aligned}$$

Accordingly $hdep(A, i'')$.

In all the cases, $\neg\, hdep(A, i'')$ implies $\boldsymbol{a}_{i''}\boldsymbol{x} = c_{i''}$. Therefore $\boldsymbol{x} \in S$. □

By this lemma, we can move any hierarchically independent row to an arbitrary upper position. For example, consider the extended coefficient matrix (5.5) again. Since the fifth row is hierarchically independent, we can elevate it and obtain any of the following matrices:

$$\left(\begin{array}{ccc|c} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 3 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 4 \\ 0 & 1 & 1 & 3 \end{array}\right), \left(\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 4 \\ 0 & 1 & 1 & 3 \end{array}\right), \left(\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 4 \\ 0 & 1 & 1 & 3 \end{array}\right), \left(\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 4 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 3 \end{array}\right).$$

All of these matrices generate the same solution set $\{(0\ 1\ 2)^{\mathrm{T}}\}$ as that of (5.5).

The row elevation operation is quite applicable to existing algorithms for solving non-hierarchical linear systems, especially ones categorized into direct methods. In the next section, using elevation, we will modify two famous direct methods called Gaussian elimination and Crout's method so that they can solve HLSs. We use the elevation lemma to prove the correctness of the resulting algorithms.

Finally, we present a convenient notion called *regularization*:

Definition 5.7 (regularization). A regularization of $(A\ \boldsymbol{c})$ is the following form of a matrix:

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & c_1 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & c_m \\ 1 & 0 & \cdots & 0 & c_{m+1} \\ 0 & 1 & & 0 & c_{m+2} \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & 1 & c_{m+n} \end{array}\right)$$

where $c_{m+1}, c_{m+2}, \ldots, c_{m+n}$ are arbitrary scalars.

Proposition 5.8. *The solution of a regularization of* $(A\ \boldsymbol{c})$ *is a solution of* $(A\ \boldsymbol{c})$*, and is unique.*
Proof. Trivial by Proposition 5.3. □

The $(m+j)$-th row of a regularization expresses $x_j = c_{m+j}$. Intuitively, it works as a default constraint that assigns c_{m+j} to x_j. Since all the variables have default values, the solution will be uniquely determined. With this property, we can simplify algorithms for solving HLSs. In the following section on algorithms, we assume that we are given only 'regularized' matrices.

5.4 Basic Algorithms

In this section, we describe algorithms for solving HLSs.

5.4.1 Design Strategy

In Lemma 5.6, we presented an operation for HLSs called elevation, which preserves the solution sets of them. With this elevation operation, we design algorithms for HLSs. To apply an elevation to an HLS, it is required that the row to elevate is hierarchically independent. Therefore, it is the key in our algorithm design to find hierarchically independent rows efficiently. We found that it is possible to efficiently realize elevation in certain algorithms belonging to *direct methods*, which solve non-hierarchical linear systems with finitely many arithmetic operations.

We actually provide three algorithms for solving HLSs. First, to illustrate how to use elevation, we show that a local propagation algorithm Blue

[58], one of the earliest algorithms for constraint hierarchies, can be rewritten in terms of HLSs using elevation. Next, we present a new algorithm based on a direct method called Gaussian elimination. Finally, we propose another algorithm that obtains LU decomposition with a direct method.

5.4.2 Local Propagation

Blue is one of the earliest algorithms for solving constraint hierarchies. It handles *multi-way constraints*, each of which consists of a set of *methods*, or functions from input variables to output variables. For example, a multi-way constraint expressing $x + y = z$ has methods $x \leftarrow z - y$, $y \leftarrow z - x$, and $z \leftarrow x + y$. To solve hierarchies of multi-way constraints, Blue chooses appropriate methods so that they will form a directed acyclic graph as a whole, leaving methods of weak unsatisfiable constraints unselected. For instance, a hierarchy with required $x + y = z$, strong $x = 1$, medium $y = 2$, and weak $z = 4$ has the set of selected methods $z \leftarrow x+y$, $x \leftarrow 1$, and $y \leftarrow 2$, where no method is chosen for the unsatisfiable constraint weak $z = 4$. After finding a method graph, Blue applies topological sort to it and executes each method in the obtained order. However, all constraint hierarchies do not have acyclic method graphs that generate correct solutions. For hierarchies with no correct acyclic method graphs, Blue finds incorrect graphs instead, that is, it is an unsound algorithm.

Originally, Blue was invented as a graph algorithm, but its background idea is not restricted to graph concepts. Therefore, we first present Blue as a simple algorithm for HLSs by modifying the one in [58].

Blue is a local propagation algorithm that can be further categorized into *propagation of known states* (PKS) [3]. Basically, PKS repeatedly selects a constraint with a method whose inputs are known. To incorporate PKS into our matrix formulation of HLSs, we view the process as transformation of coefficient matrices into lower triangular ones.[5] Formally, to solve an HLS

[5] The other category is *propagation of degrees of freedom* (PDF) [3], which, at each step, chooses a constraint with a method whose output variable can be arbitrarily determined. With PDF, coefficient matrices would result in upper triangular ones.

$A\boldsymbol{x} = \boldsymbol{c}$, where A is a regularization, we transform $(A\ \boldsymbol{c})$ into

$$(B\ \boldsymbol{d}) = \left(\begin{array}{cccc|c} b_{11} & 0 & \cdots & 0 & d_1 \\ b_{21} & b_{22} & & 0 & d_2 \\ \vdots & \vdots & \ddots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} & d_n \\ b_{n+1,1} & b_{n+1,2} & \cdots & b_{n+1,n} & d_{n+1} \\ \vdots & \vdots & & \vdots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,n} & d_m \end{array}\right),$$

where $b_{jj} \neq 0$ for $1 \leq j \leq n$, by applying a sequence of row elevation operations and column swap operations. Note that, when we apply a column swap operation, we must also swap the corresponding variables, and therefore, we finally obtain a permutation $\boldsymbol{y}$ of $\boldsymbol{x}$. In $(B\ \boldsymbol{d})$, the first to n-th rows are hierarchically independent, which is always possible because A is a regularization. Accordingly, we can obtain the solution of $B\boldsymbol{y} = \boldsymbol{d}$ by computing $y_j \leftarrow 1/b_{jj}(d_j - \sum_{j'=1}^{j-1} b_{jj'}y_{j'})$ for $j = 1, 2, \ldots, n$ in this order, which is known as forward substitution. We can regard such a transformation as PKS, where each assignment to y_j works as a selected method.

Now we present the Blue algorithm modified for HLSs:

```
 1 blue((A c))
 2 {
 3   m ← # of rows of A; n ← # of columns of A;
 4   for j ← 1 to n do {
 5     for i ← j to m do { // find a hierarchically independent row.
 6       j_o ← 0;
 7       for j' ← j to n do
 8         if a_ij' ≠ 0 then { j_o ← j'; break; }
 9       for j' ← j' + 1 to n do
10         if a_ij' ≠ 0 then { j_o ← 0; break; }
11       if j_o ≠ 0 then break;
12     }
13     if i ≠ j then elev_row((A c), i, j); // elevate the row.
14     if j_o ≠ j then swap_cols(A, j_o, j); // also variables.
15   }
16 }
```

The algorithm `blue` takes $(A\ \boldsymbol{c})$ as input, and then rewrite it into a lower triangular matrix. `elev_row`$((A\ \boldsymbol{c}), i, j)$ applies a row elevation operation to

$(A\ \boldsymbol{c})$ by elevating the i-th row to the j-th, and `swap_cols`(A, j_o, j) performs a column swap operation by exchanging the j_o-th and j-th columns of A.

Intuitively, `blue` iterates the `for` j loop from line 3 to select a uniquely satisfiable equation. Before the j-th step, $(A\ \boldsymbol{c})$ has been rewritten into the following form:

$$\left(\begin{array}{ccc|ccc|c} b_{11} & & 0 & 0 & \cdots & 0 & d_1 \\ \vdots & \ddots & & \vdots & & \vdots & \vdots \\ b_{j-1,1} & \cdots & b_{j-1,j-1} & 0 & \cdots & 0 & d_{j-1} \\ \hline a'_{j1} & \cdots & a'_{j,j-1} & a'_{jj} & \cdots & a'_{jn} & c'_j \\ \vdots & & \vdots & \vdots & & \vdots & \vdots \\ a'_{m1} & \cdots & a'_{m,j-1} & a'_{mj} & \cdots & a'_{mn} & c'_m \end{array}\right).$$

By this step, the rows over the j-th have been processed, and partially form a lower triangular matrix. Then, at lines 5–12, the algorithm searches for the uppermost unprocessed row with exactly one nonzero at the j-th to n-th columns, that is, if it is the i-th row, it has the form $(a_{i1}\ a_{i2}\ \cdots\ a_{i,j-1}\ 0\ \cdots\ 0\ a_{ij_o}\ 0\ \cdots\ 0)$, where $a_{ij_o} \neq 0$. At lines 13 and 14, it moves a_{ij_o} to the (j, j) position by elevating the i-th row to the j-th and swapping the j_o-th and j-th columns.

To illustrate the correctness of the algorithm, we show that it chooses only hierarchically independent rows to elevate. Since Blue is inherently unsound, we make the following premise for the correctness: we do not need to simultaneously satisfy two or more equations to obtain variable values. Let the i-th row be selected for elevation, and $\boldsymbol{a}_i = (a_{i1}\ \cdots\ a_{i,j-1}\ 0\ \cdots\ 0\ a_{ij_o}\ 0\ \cdots\ 0)$. Assume, for contradiction, that it is hierarchically dependent. Then there exist some $\alpha_{i_1}, \ldots, \alpha_{i_l}$ such that $\boldsymbol{a}_i = \alpha_{i_1}\boldsymbol{a}_{i_1} + \cdots + \alpha_{i_l}\boldsymbol{a}_{i_l}$, where $i_1, \ldots, i_l < i$ and $\boldsymbol{a}_{i_1}, \ldots, \boldsymbol{a}_{i_l}$ are independent. It follows $(\alpha_{i_1}a_{i_1j_o} + \cdots + \alpha_{i_l}a_{i_lj_o})x_{j_o} = \alpha_{i_1}c_{i_1} + \cdots + \alpha_{i_l}c_{i_l}$, and therefore x_{j_o} is uniquely determined. However, since the rows over the j-th have only zeros at the j-th to n-th columns, we must have at least two i_k's such that $i_k \geq j$. Thus, we must simultaneously satisfy at least two equations to obtain the value of x_{j_o}, which is contradiction to our premise. Therefore, the i-th row is hierarchically independent, and the row elevation operation is valid.

The time complexity of the Blue algorithm for HLSs is $O(mn^2)$. It is slower than the original Blue algorithm because of the dense matrix notation of HLSs instead of graphs.

5.4.3 Elimination

Next, we present a new algorithm for solving HLSs that is sound unlike Blue. We based this algorithm on Gaussian elimination, which is one of the most familiar direct-method algorithms for solving non-hierarchical linear systems. Similar to Gaussian elimination, our algorithm transforms coefficient matrices into upper triangular ones. Formally, given $(A\ \boldsymbol{c})$, where A is a regularization, it outputs

$$(B\ \boldsymbol{d}) = \left(\begin{array}{cccc|c} b_{11} & b_{12} & \cdots & b_{1n} & d_1 \\ 0 & b_{22} & \cdots & b_{2n} & d_2 \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & & b_{nn} & d_n \\ 0 & 0 & \cdots & 0 & d_{n+1} \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & d_m \end{array}\right),$$

where $b_{jj} \neq 0$ for $1 \leq j \leq n$. Since the first to n-th rows of $(B\ \boldsymbol{d})$ are hierarchically independent, we can obtain the solution of $B\boldsymbol{x} = \boldsymbol{d}$ by calculating $x_j \leftarrow 1/b_{jj}(d_j - \sum_{j'=j+1}^{n} b_{jj'}x_{j'})$ for $j = n, n-1, \ldots, 1$ in this order, which is known as backward substitution.

Below is our elimination algorithm for HLSs:

```
 1 elim((A c))
 2 {
 3   m ← # of rows of A; n ← # of columns of A;
 4   for j ← 1 to n do {
 5     for i ← j to m do // find a hierarchically independent row.
 6       if a_ij ≠ 0 then break;
 7     if i ≠ j then elev_row((A c), i, j); // elevate the row.
 8     for i ← j + 1 to m do {
 9       r ← a_ij/a_jj;
10       for j' ← j to n do a_ij' ← a_ij' − r * a_jj';
11       c_i ← c_i − r * c_j;
12     }
13   }
14 }
```

Before the j-th step of the `for` j loop from 3, $(A\ \boldsymbol{c})$ has been transformed

into the following form:

$$\left(\begin{array}{ccc|ccc|c} b_{11} & \cdots & b_{1,j-1} & b_{1j} & \cdots & b_{1n} & d_1 \\ & \ddots & \vdots & \vdots & & \vdots & \vdots \\ 0 & & b_{j-1,j-1} & b_{j-1,j} & \cdots & b_{j-1,n} & d_{j-1} \\ \hline 0 & \cdots & 0 & a'_{jj} & \cdots & a'_{jn} & c'_j \\ \vdots & & \vdots & \vdots & & \vdots & \vdots \\ 0 & \cdots & 0 & a'_{mj} & \cdots & a'_{mn} & c'_m \end{array}\right).$$

Intuitively, at lines 4–6, the algorithm looks for the uppermost unprocessed row with a nonzero at the j-th column. Next, at line 7, it elevates the selected i-th row to the j-th. Finally, at lines 8 to 12, it 'eliminates' all the lower entries at the j-th column with the new j-th row in the same way as Gaussian elimination.

Now we show that this algorithm chooses only hierarchically independent rows to elevate. Let the i-th row be selected. Assume, for contradiction, that the i-th row is dependent. Then there exist some $\alpha_1, \ldots, \alpha_{i-1}$ such that $\boldsymbol{a}_i = \alpha_1 \boldsymbol{a}_1 + \cdots + \alpha_{i-1}\boldsymbol{a}_{i-1}$. Since $a_{11} \neq 0$ and $a_{i'1} = 0$ for $i' > 1$, $\alpha_1 = 0$ is required. Inductively, $\alpha_{i'} = 0$ is also required for $i' < j$. Hence $\boldsymbol{a}_i = \alpha_j \boldsymbol{a}_j + \cdots + \alpha_{i-1}\boldsymbol{a}_{i-1}$, which is contradiction because $a_{i'j} = 0$ for $j \leq i' < i$ but $a_{ij} \neq 0$. Accordingly, the i-th row is independent.

We can accommodate this algorithm to 'unregularized' coefficient matrices. If we give an unregularized matrix to it, it would possibly encounter the situation that $a_{ij} = 0$ for all $i \geq j$ at lines 5 and 6. To cope with such a situation, it would need to exchange the j-th column with another remaining column that have at least one nonzero. With such modification, the algorithm would finally obtain a set of equations that represents a set of all solutions.

The time complexity of our elimination algorithm is $O(mn^2)$, which is the same as that of the Blue algorithm for HLSs. It means that our elimination algorithm obtains soundness and completeness without increasing time complexity.

5.4.4 LU Decomposition

We present another sound algorithm based on Crout's method, a direct-method algorithm for applying LU decomposition to square matrices. It transforms a coefficient matrix into a product of lower and upper triangular ones. Formally, from a regularization A, it obtains the product of two

matrices L and U as follows:

$$LU = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & & 0 \\ \vdots & & \ddots & \\ l_{n1} & l_{n2} & & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & & u_{nn} \end{pmatrix},$$

where $u_{jj} \neq 0$ for $1 \leq j \leq n$. The advantage of this algorithm over the previous elimination method is that we can compute solutions of $A\boldsymbol{x} = \boldsymbol{c}$ for different $\boldsymbol{c}$ without recomputing L and U. Concretely, once we created L and U from A, we can solve $A\boldsymbol{x} = \boldsymbol{c}$ as follows: first, obtain $\boldsymbol{d}$ by selecting entries from $\boldsymbol{c}$ in the way determined at LU decomposition; next, solve $L\boldsymbol{y} = \boldsymbol{d}$ for $\boldsymbol{y}$ with forward substitution; finally, solve $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$ with backward substitution. This property is useful for GUIs because it allows us to realize two-phase constraint satisfaction similar to local propagation.

Now we present the algorithm below:

```
 1 lu_dec(A)
 2 {
 3   m ← # of rows of A; n ← # of columns of A;
 4   for j ← 1 to n do {
 5     for i ← 1 to j − 1 do
 6       for k ← 1 to i − 1 do a_ij ← a_ij − a_ik * a_kj;
 7     for i ← j to m do
 8       for k ← 1 to j − 1 do a_ij ← a_ij − a_ik * a_kj;
 9     for i ← j to m do // find a hierarchically independent row.
10       if a_ij ≠ 0 then break;
11     if i ≠ j then elev_row(A, i, j); // elevate the row.
12     for i ← j + 1 to m do a_ij ← a_ij/a_jj;
13   }
14 }
```

It overwrites entries of L and U on A. Before the j-th step of the `for` j loop from line 4, A has been transformed into

$$\left(\begin{array}{cccc|ccc} u_{11} & u_{12} & \cdots & u_{1,j-1} & a'_{1j} & \cdots & a'_{1n} \\ l_{21} & u_{22} & \cdots & u_{2,j-1} & a'_{2j} & \cdots & a'_{2n} \\ \vdots & & \ddots & \vdots & \vdots & & \vdots \\ l_{j-1,1} & l_{j-1,2} & & u_{j-1,j-1} & a'_{j-1,j} & \cdots & a'_{j-1,n} \\ \hline l'_{j1} & l'_{j2} & \cdots & l'_{j,j-1} & a'_{jj} & \cdots & a'_{jn} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ l'_{m1} & l'_{m2} & \cdots & l'_{m,j-1} & a'_{mj} & \cdots & a'_{mn} \end{array}\right).$$

We informally show why the i-th row selected at lines 9 and 10 is hierarchically independent. At this time, for $1 \leq i' < j$, all

$$\boldsymbol{b}_{i'}^{(j)} = l_{i'1}\boldsymbol{u}_1^{(j)} + \cdots + l_{i',i'-1}\boldsymbol{u}_{i'-1}^{(j)} + \boldsymbol{u}_{i'}^{(j)}$$

are linearly independent, where $l_{i'k} = a_{i'k}$ and $\boldsymbol{u}_k^{(j)} = (0 \ \cdots \ 0 \ u_{kk} \ \cdots u_{kj}) = (0 \ \cdots \ 0 \ a_{kk} \ \cdots a_{kj})$ (each a_{st} represents the current value of the (s,t) entry). It indicates that $\boldsymbol{u}_{i'}^{(j)}$ can be uniquely expressed with $\boldsymbol{b}_1^{(j)}, \ldots, \boldsymbol{b}_{i'}^{(j)}$ as follows:

$$\boldsymbol{u}_{i'}^{(j)} = \beta_{i'1}\boldsymbol{b}_1^{(j)} + \cdots + \beta_{i'i'}\boldsymbol{b}_{i'}^{(j)}. \tag{5.6}$$

For $j \leq i' < i$, the following equation holds since $a_{i'j} = 0$:

$$\boldsymbol{b}_{i'}^{(j)} = l_{i'1}\boldsymbol{u}_1^{(j)} + l_{i'2}\boldsymbol{u}_1^{(j)} + \cdots + l_{i',j-1}\boldsymbol{u}_{j-1}^{(j)},$$

By (5.6), it follows that $\boldsymbol{b}_{i'}^{(j)}$ is dependent on $\boldsymbol{b}_1^{(j)}, \ldots, \boldsymbol{b}_{j-1}^{(j)}$. By contrast, we have

$$\boldsymbol{b}_i^{(j)} = l_{i1}\boldsymbol{u}_1^{(j)} + l_{i2}\boldsymbol{u}_1^{(j)} + \cdots + l_{i,j-1}\boldsymbol{u}_{j-1}^{(j)} + (0 \ \cdots \ 0 \ a_{ij}), \tag{5.7}$$

where $a_{ij} = 0$. However, ignoring the j-th column, we also have

$$\boldsymbol{b}_i^{(j-1)} = l_{i1}\boldsymbol{u}_1^{(j-1)} + l_{i2}\boldsymbol{u}_1^{(j-1)} + \cdots + l_{i,j-1}\boldsymbol{u}_{j-1}^{(j-1)}.$$

Since $\boldsymbol{u}_1^{(j-1)}, \ldots, \boldsymbol{u}_{j-1}^{(j-1)}$ are linearly independent, $l_{i1}, \ldots, l_{i,j-1}$ must be unique. Therefore, we have only one combination of $l_{i1}, \ldots, l_{i,j-1}$ to equate the first to $(j-1)$-th entries in (5.7), but in this case we cannot equate the j-th entries without a_{ij}. It follows that $\boldsymbol{b}_i^{(j)}$ is linearly independent of $\boldsymbol{b}_1^{(j)}, \ldots, \boldsymbol{b}_{j-1}^{(j)}$. Therefore, $\boldsymbol{b}_i^{(j)}$ is hierarchically independent.

As is with the previous elimination algorithm, we can also adapt this algorithm to unregularized coefficient matrices. If we supply an unregularized matrix to it, it would possibly find that $a_{ij} = 0$ for all $i \geq j$ at lines 9 and 10. Therefore, it is necessary to swap the column with another remaining column with one or more non-zeros. Finally, it would acquire a set of equations indicating the solution set.

The time complexity of our LU decomposition algorithm is $O(mn^2)$, which is equal to that of the previous elimination algorithm. However, as we noted, the LU decomposition algorithm has an extra advantage that it efficiently recomputes solutions of $\boldsymbol{Ax} = \boldsymbol{c}$ for different $\boldsymbol{c}$.

5.5 Discussion

This section provides minor discussions on HLSs.

5.5.1 Limitations Owing to Total Ordering of Preferential Constraints

We sometimes encounter problems due to total ordering of preferential constraints. A typical example is that we have difficulty in dragging an object constrained to be on a almost horizontal or vertical straight line. In this case, we cannot simultaneously satisfy both of the two constraints that bind the mouse cursor with the object (one for x-coordinates and the other for y-coordinates). Since preferential constraints are totally ordered, the stronger one of the binding constraints will be selected by the solver. Suppose that the binding constraint for x-coordinates is stronger, and also that the line is almost vertical. Then, if the user moves the mouse slightly in its x-coordinate, the object will be largely moved in its y-coordinate. It is because the almost vertical line determines the y-coordinate by largely magnifying the change of the x-coordinate.[6]

We can alleviate this problem to a certain degree by introducing a hybrid comparator, which we discuss in the next subsection.

5.5.2 Hybrid Comparators

Our theory and algorithms are restricted to locally-better. However, in some situation, we can accommodate our algorithms to a hybrid comparator, for example, that uses locally-predicate-better (LPB) for some levels and also least-squares-better (LSB) for the other levels.

Consider that we solve a constraint hierarchy consisting of linear equations with a hybrid comparator that adopts LPB for levels 1 to $n-1$ and LSB for level n. By re-ordering constraints at levels 1 to $n-1$ within each level, we can obtain an HLS $A\boldsymbol{x} = \boldsymbol{c}$. When we solve it without regularization, we acquire $\boldsymbol{y} = A'\boldsymbol{z} + \boldsymbol{c}'$, where $\boldsymbol{y}$ and $\boldsymbol{z}$ contain distinct variables formerly in $\boldsymbol{x}$. Next, we integrate it with the linear system $B\boldsymbol{x} = \boldsymbol{d}$ for level n by eliminating $\boldsymbol{y}$ with $\boldsymbol{y} = A'\boldsymbol{z} + \boldsymbol{c}'$, and then get $B'\boldsymbol{z} = \boldsymbol{d}'$. Now, we can obtain the solution of the original hierarchy by solving $B'\boldsymbol{z} = \boldsymbol{d}'$ with a known algorithm for the least-squares method of linear systems.

[6] Note that this is not a unique problem that we encounter by using HLSs. In fact, HLSs inherit this problem from constraint hierarchies solved with locally-better. Also, remind that, in Chapter 4, we designed the *DETAIL* constraint solver to tackle this problem in the framework of local propagation.

5.5.3 Pivoting

As we showed, finding hierarchically independent rows can be efficiently realized in certain direct-method algorithms. At the positions where we inserted row elevation operations, the original algorithms may perform *partial pivoting*, which swaps a row with another lower row that causes computation errors to increase in the least degree. Concretely, both Gaussian elimination and Crout's method seek the rows with the largest values, instead of the uppermost rows with non-zeros as is with our algorithms. It suggests that, by replacing partial pivoting with elevation, we may be able to accommodate various direct methods to hierarchies.

It also indicates that we cannot use partial pivoting to prevent computation errors. A simple solution is to use *complete pivoting*, which exchanges columns for this purpose. However, complete pivoting is usually more expensive than partial pivoting. We can regard it as the additional cost because of introducing hierarchies. Currently, we ourselves are not sensitive to computation errors, because we mainly use constraints for graphical user interfaces.

Chapter 6

The HiRise Constraint Solver

This chapter describes the HiRise constraint solver, which provides incremental planning and real-time execution for satisfying HLSs.

6.1 Overview

HiRise is an incremental algorithm for solving HLSs consisting of required and preferential linear equality constraints. To allow incremental addition and removal of constraints, it 'factorizes' coefficient matrices into products of certain matrices that enable us to easily replace rows of original matrices.

We present the following two strategies for incremental satisfaction of HLSs:

- The basic strategy is for required constraints. Since all required constraints must be satisfied by definition, they must also be satisfied after a constraint is added or removed. Therefore, using the advantage of the matrix maintenance mechanism of HiRise, it can easily update data structures for required constraints. After that, it re-solves preferential constraints non-incrementally. This strategy is expected to be efficient for HLSs with much fewer preferential constraints than required ones.

- The enhanced strategy is for preferential constraints. Preferential constraints are more difficult to incrementally handle than required ones, because a satisfied preferential constraint may become unsatisfied due

to addition of another stronger constraint. Therefore, the key to incremental treatment of preferential constraints is to quickly find a constraint that should be influenced by addition of a constraint. HiRise realizes this by introducing walkabout strengths in a similar way to *DETAIL*.

6.2 Algorithm

This section describes the constraint satisfaction algorithm used in the HiRise constraint solver.

6.2.1 Non-Incremental Satisfaction of HLSs

This subsection presents the algorithm that satisfies HLSs non-incrementally. A typical situation that HiRise uses this algorithm is that it solves an initial HLS just after an application is started.

In the following description, we first treat HLSs consisting only of preferential constraints for simplicity. We will mention how to handle required constraints later in this subsection.

Triangular Factorization

In the planning phase, the non-incremental algorithm chooses hierarchically independent rows from the coefficient matrix of a given HLS, and then applies *triangular factorization* to the selected rows. To illustrate triangular factorization, assume that hierarchically independent rows $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, ..., $\boldsymbol{b}_n$ are already collected and form an $n \times n$ matrix $B = \begin{pmatrix} \vdots \end{pmatrix}$. Then we can express the triangular factorization of B as follows:

$$BP_1U_1P_2U_2 \cdots P_nU_n = L \tag{6.1}$$

where each P_i is an $n \times n$ permutation matrix such that

$$P_i = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & & & 1 & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & 1 & & & 0 & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \leftarrow i\text{-th} \\ \\ \\ \\ \leftarrow j\text{-th} \\ \\ \\ \end{matrix}$$

each U_i is an $n \times n$ upper triangular eta matrix[1] such that

$$U_i = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & \frac{1}{b'_{ii}} & -\frac{b'_{i,i+1}}{b'_{ii}} & \cdots & -\frac{b'_{in}}{b'_{ii}} & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & & & & 1 & \end{pmatrix} \leftarrow i\text{-th} \tag{6.2}$$

and L is an $n \times n$ lower triangular matrix such that

$$L = \begin{pmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ \vdots & \ddots & \ddots & & \\ l_{n-1,1} & l_{n-1,2} & \ddots & 1 & \\ l_{n1} & l_{n2} & \cdots & l_{n,n-1} & 1 \end{pmatrix}.$$

Intuitively, the sequence P_1, U_1, P_2, U_2, ..., P_n, U_n transforms B into a lower triangular matrix L. Before multiplied by the i-th pair of matrices

[1] Normally, except diagonal entries, an eta matrix has non-zero entries at a column. By contrast, U_i contains non-zero entries at the i-th row. However, for brevity, we also refer U_i as an eta matrix.

P_i and U_i, B has been transformed as follows:

$$
\begin{aligned}
&BP_1U_1\cdots P_{i-1}U_{i-1} \\
&= \begin{pmatrix}
1 & & & & & & & & \\
b'_{21} & 1 & & & & & & & \\
\vdots & \ddots & \ddots & & & & & & \\
\vdots & & \ddots & \ddots & & & & & \\
b'_{i-1,1} & b'_{i-1,2} & \cdots & b'_{i-1,i-2} & 1 & & & & \\
b'_{i1} & b'_{i2} & \cdots & b'_{i,i-2} & b'_{i,i-1} & b'_{ii} & b'_{i,i+1} & \cdots & b'_{in} \\
b'_{i+1,1} & b'_{i+1,2} & \cdots & b'_{i+1,i-2} & b'_{i+1,i-1} & b'_{i+1,i} & b'_{i+1,i+1} & \cdots & b'_{i+1,n} \\
\vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots \\
b'_{n1} & b'_{n2} & \cdots & b'_{n,i-2} & b'_{n,i-1} & b'_{ni} & b'_{n,i+1} & \cdots & b'_{nn}
\end{pmatrix}.
\end{aligned}
$$

P_i swaps the i-th column for the j-th so that the resulting (i,i)-entry is not 0. Then U_i changes the (i,i)-entry into 1, and 'eliminates' all (i,j)-entries for $j > i$. At the same time, U_i modifies lower entries at the i-th to n-th columns.

Our triangular factorization may be easily understood when compared with Gaussian elimination: it transforms a square matrix B into a lower triangular matrix L, whereas Gaussian elimination transforms a square matrix into an upper triangular matrix. Also, it is different from Gaussian elimination in that it records its transformation process as a sequence of P_i's and U_i's, which we call *transformation matrices*.

The Planning Phase

Below, we present the pseudo-code of the non-incremental algorithm for obtaining the triangular factorization of a coefficient matrix A:

```
1  j ← 1;
2  for i ← 1 to m do {
3     (a_1 a_2 ··· a_n) ← a_i P_1 U_1 P_2 U_2 ··· P_{j-1} U_{j-1};
4     if ∃j' ≥ j. a_{j'} ≠ 0 then {
5        b_j ← a_i;
6        determine P_j and U_j;
7        l_j ← (a_1 a_2 ··· a_n) P_j U_j;
8        j ← j + 1;
9     }
10 }
```

where $\boldsymbol{b}_j$ and $\boldsymbol{l}_j$ are the j-th rows of B and L respectively.

Intuitively, in the `for` loop from line 2, it checks whether the i-th row $\boldsymbol{a}_i$ of A is hierarchically independent, and if the row is independent, it decides the j-th pair of P_j and U_j. Specifically, at line 3, it transforms $\boldsymbol{a}_i$ into $(a_1\ a_2\ \cdots\ a_n)$ by applying already determined P_k's and U_k's. Then, at line 4, it examines whether the row is hierarchically independent: if all $a_{j'}$ for $j' \geq j$ are zero, $\boldsymbol{a}_i$ is dependent on rows already judged hierarchically independent; otherwise, $\boldsymbol{a}_i$ is hierarchically independent. If $\boldsymbol{a}_i$ is independent, it assigns $\boldsymbol{a}_i$ to $\boldsymbol{b}_j$ at line 5, and it determines P_j and U_j at line 6, based on triangular factorization explained earlier. At line 7, it computes the j-th row $\boldsymbol{l}_j$ of the lower triangular matrix L, and then finishes this step with increasing j at line 8.

The Execution Phase

The execution phase solves $B\boldsymbol{x} = \boldsymbol{c}$ for $\boldsymbol{x}$, which is as simple as substitution operations for ordinary LU decomposition.

By introducing an n-vector $\boldsymbol{y}$, let

$$\boldsymbol{x} = P_1U_1P_2U_2\cdots P_nU_n\boldsymbol{y}. \tag{6.3}$$

Then $L\boldsymbol{y} = \boldsymbol{c}$ holds since

$$\begin{aligned} L\boldsymbol{y} &= BP_1U_1\cdots P_nU_n\boldsymbol{y} \\ &= BP_1U_1\cdots P_nU_n(P_1U_1\cdots P_nU_n)^{-1}\boldsymbol{x} \\ &= B\boldsymbol{x} \\ &= \boldsymbol{c}. \end{aligned}$$

The execution phase first solves $L\boldsymbol{y} = \boldsymbol{c}$ for $\boldsymbol{y}$ with forward substitution. Then it computes $\boldsymbol{x}$ with (6.3), which is easy because of the simple structures of P_i's and U_i's.

Treating Required Constraints

Required constraints differ from preferential constraints in that they must be always satisfied by definition. We can realize required constraints by slightly modifying the algorithm described above. Basically, we handle required constraints by locating them on top of preferential ones in coefficient matrices. Furthermore, to guarantee all of them to hold, the run-time solver records required constraints proved to be hierarchically dependent in the planning phase, and checks whether all the recorded ones are satisfied every time after the execution phase. If the solver detects an unsatisfied constraint, it will report an error to its application.

6.2.2 Modifying Triangular Factorizations

Before describing incremental satisfaction of HLSs, we present its basic technique.

Technique

By modifying the triangular factorization of an $n \times n$ matrix B, it obtains a factorization of a matrix B' revised by eliminating a row from B and then adding a new row to the bottom instead, i.e.

$$B' = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ \vdots & \vdots & & \vdots \\ b_{i-1,1} & b_{i-1,2} & \cdots & b_{i-1,n} \\ b_{i+1,1} & b_{i+1,2} & \cdots & b_{i+1,n} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \\ b'_{i1} & b'_{i2} & \cdots & b'_{in} \end{pmatrix}.$$

Using the triangular factorization (6.1) of B, it obtains the following form of a factorization:

$$B' P_1 U_1 P_2 U_2 \cdots P_n U_n Q U'_i U'_{i+1} \cdots U'_n = L'$$

where all P_k and U_k are the same as those of (6.1), Q is an $n \times n$ permutation matrix such that

$$Q = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 0 & & 1 & \\ & & & 1 & & & \\ & & & & \ddots & & \\ & & & & & 1 & 0 \end{pmatrix} \leftarrow i\text{-th}$$

and each U_i is an $n \times n$ upper triangular eta matrix of the form (6.2).

First, in this factorization, multiplying B' by $P_1 U_1 P_2 U_2 \cdots P_n U_n$ obtains

the following:

$$
\begin{aligned}
&B'P_1U_1P_2U_2\cdots P_nU_n \\
&= \begin{pmatrix}
1 & & & & & & \\
\vdots & \ddots & & & & & \\
l_{i-1,1} & & 1 & & & & \\
l_{i+1,1} & \cdots & l_{i+1,i-1} & l_{i+1,i} & 1 & & \\
\vdots & & \vdots & \vdots & & \ddots & \\
l_{n1} & \cdots & l_{n,i-1} & l_{ni} & l_{n,i+1} & & 1 \\
b''_{i1} & \cdots & b''_{i,i-1} & b''_{ii} & b''_{i,i+1} & & b'_{in}
\end{pmatrix}.
\end{aligned}
$$

Note that the first to $(i-1)$-th rows are the same as those of the lower triangular matrix L from the factorization of B respectively, and that the i-th to $(n-1)$-th rows are the same as the $(i+1)$-th to n-th rows of L respectively. Next, multiplying $B'P_1U_1P_2U_2\cdots P_nU_n$ by Q results in

$$
\begin{aligned}
&B'P_1U_1P_2U_2\cdots P_nU_nQ \\
&= \begin{pmatrix}
1 & & & & & & \\
\vdots & \ddots & & & & & \\
l_{i-1,1} & & 1 & & & & \\
l_{i+1,1} & \cdots & l_{i+1,i-1} & 1 & & & l_{i+1,i} \\
\vdots & & \vdots & & \ddots & & \vdots \\
l_{n1} & \cdots & l_{n,i-1} & l_{n,i+1} & & 1 & l_{ni} \\
b''_{i1} & \cdots & b''_{i,i-1} & b''_{i,i+1} & & b''_{in} & b''_{ii}
\end{pmatrix}.
\end{aligned}
$$

Now, we can easily acquire a lower triangular matrix L' by eliminating each (k,n)-entry for $i \le k < n$ with U'_k and also changing the (n,n)-entry into 1 with U'_n.

Clearly, we can repeatedly perform this technique, because we can further apply it to a factorization that it previously obtained. When it is repeated, the sequence of upper triangular eta matrices such as U'_k are extended. However, for brevity, we also refer such a resulting factorization as a triangular factorization.

This technique is closely related to Forrest and Tomlin's method known in linear programming [16], which was developed for substituting columns of matrices. Forrest and Tomlin's method handles triangular factorization of the form

$$L_nP_nL_{n-1}P_{n-1}\cdots L_1P_1B = U$$

which can be regarded as a variation of Gaussian elimination that records its transformation process. Forrest and Tomlin's method modifies the triangular factorization by multiplying it by $L'_n L'_{n-1} \cdots L'_i Q$ from the left.

Re-Factorization

The technique for modifying triangular factorizations suggests that a constraint solver using this technique will become slower as it is repeated, because the sequence of transformation matrices gets longer. In fact, both of the planning and execution phases in the incremental algorithm take longer times as the sequence extends. To prevent the constraint solver from getting slow, it is useful to shorten the sequence by performing the non-incremental algorithm occasionally. We refer such an interleaved non-incremental factorization as *re-factorization*.[2]

6.2.3 Incremental Maintenance of Required Constraints

This subsection describes the basic strategy for incremental constraint satisfaction. As described, we incrementally maintain only required constraints with this strategy.

Adding a Required Constraint

First, we provide how to add a required constraint $\boldsymbol{a}^0_{m_0+1}\boldsymbol{x} = c^0_{m_0+1}$ to an HLS $\langle (A_0\ \boldsymbol{c}_0), (A\ \boldsymbol{c}) \rangle$ that is already factorized.

For simplicity, we assume that all existing required constraints are independent. Then we do not need to change P_i, U_i, and $\boldsymbol{l}_i$ for $i \leq m_0$; since required constraints must be satisfied by definition, we can skip the factorization for them. Therefore, for required constraints, we only need to factorize $\boldsymbol{a}^0_{m_0+1}$. If $\boldsymbol{a}^0_{m_0+1}$ is hierarchically dependent, we also do not have to change the remaining P_i's, U_i's, and $\boldsymbol{l}_i$'s (in this case, the satisfiability of the added constraint must be examined for each substitution). If $\boldsymbol{a}^0_{m_0+1}$ is hierarchically independent, we factorize $\boldsymbol{a}^0_{m_0+1}$ and also the rows of preferential constraints in the same way as non-incremental satisfaction of HLSs.

[2] The appropriate frequency for re-factorization depends on actual applications. In linear programming, some report says that a good ratio is one re-factorization to twenty incremental factorizations [16].

Removing a Required Constraint

Next, we present the way to remove an existing required constraint $\boldsymbol{a}_i^0 \boldsymbol{x} = c_i^0$ from an HLS $\langle (A_0\ \boldsymbol{c}_0), (A\ \boldsymbol{c}) \rangle$ that is already factorized.

We assume that all existing required constraints are independent. Then $\boldsymbol{a}_i^0$ is used for factorization of the HLS. Using the technique for modifying triangular factorizations, we can incrementally factorize the remaining required constraints after deleting $\boldsymbol{a}_i^0$. Then we can factorize the rows of preferential constraints in a similar way to non-incremental satisfaction.

Adding or Removing a Preferential Constraint

The basic strategy does not support incremental maintenance of preferential constraints. Therefore, to add or remove a preferential constraint, we need to factorize the rows of all preferential constraints.

6.2.4 Incremental Maintenance of Preferential Constraints

This subsection gives the enhanced strategy, which incrementally maintains preferential constraints.

We assume that before the algorithm begins, there exists an HLS that are already solved with triangular factorization. Then, by modifying its factorization, the algorithm solves another HLS that can be obtained by inserting a constraint to the existing HLS or removing one from the HLS.

In the same way as the non-incremental algorithm, for simplicity, this subsection considers HLSs that contain only preferential constraints.

Adding a Constraint to an HLS

Now, we explain how to add a new constraint to an HLS whose triangular factorization is already obtained. By "adding a constraint to an HLS," we mean that we obtain another HLS by inserting the constraint between the $(i-1)$-th and i-th constraints of the original HLS for some i.

When we add a constraint to an HLS, we encounter either of the following two cases:

- The new constraint should be enforced; that is, we use its coefficient vector to obtain a new triangular factorization. To realize it, we revoke an appropriate enforced weaker constraint by removing the corresponding row from the matrix B, and add the coefficient vector of the new constraint to the bottom instead. Then we incrementally compute the new factorization.

- The new constraint should not be enforced. In this case, we do not need to modify the current triangular factorization.

Note that, because of the first case, the order of rows in B may become different from that of the corresponding constraints in the original HLS, whereas the non-incremental algorithm and the incremental algorithm using the basic strategy preserve the order of constraints. In spite of such change of the order, we can obtain correct solutions if all the selected constraints are originally hierarchically independent.

Now, we outline the algorithm for obtaining the triangular factorization of the new HLS:

1. Predict a constraint that will need to be revoked by adding the new constraint. If there is not such a constraint, the algorithm terminates without modifying the triangular factorization. Otherwise, let k be the row index of the predicted constraint in B.

2. Compute $(a_1\ a_2\ \cdots\ a_n)$ by multiplying the coefficient vector of the new constraint by the sequence of transformation matrices $P_1U_1P_2U_2\cdots$.

   (a) If a_k is not zero, incrementally obtain the triangular factorization of the matrix acquired by deleting the k-th row from B and adding the coefficient vector of the new constraint to the bottom.

   (b) Otherwise, switch to the basic strategy for incremental factorization.[3]

The key of this algorithm is the prediction of a constraint to revoke. For this purpose, we use walkabout strengths again in a similar way to classical local propagation algorithms such as DeltaBlue. In the HiRise algorithm, we assign a walkabout strength w_j to each variable x_j, and compute walkabout strengths as follows:

```
1  for i ← 1 to n do {
2    w_i ← index of the constraint corresponding to l_i;
3    for j ← 1 to i − 1 do if l_ij ≠ 0 then w_i ← min(w_i, w_j);
4  }
```

[3]Precisely, the process to follow is not exactly the same as the basic strategy, which was described in the previous subsection. If the basic strategy is used alone, all required constraints appear over preferential ones in B. However, when the enhanced strategy is adopted, one or more required constraints may mingle with preferential ones, because they have been added incrementally. In this case, the algorithm cannot incrementally maintain such required constraints, as opposed to the original basic strategy.

```
 5  for k ← # of transformation matrices to 1 step −1 do {
 6    let T_k be the k-th transformation matrix;
 7    if T_k is of form P then {
 8      let j and j' be the indices of the columns
           to be swapped by T_k;
 9      w ← w_j; w_j ← w_j'; w_j' ← w;
10    }
11    else if T_k is of form Q then {
12      let j be the index of the column
           to be moved to the rightmost by T_k;
13      w ← w_n;
14      for j' ← n to j + 1 step −1 do w_j' ← w_j'−1;
15      w_j ← w;
16    }
17    else { // T_k is of form U
18      let i be the index of the row to be modified by T_k;
19      for j ← i + 1 to n do if t^k_ij ≠ 0 then w_i ← min(w_i, w_j);
20    }
21  }
```

where l_{ij} is the j-th entry of $\boldsymbol{l}_i$, and t^k_{ij} is the (i,j)-entry of T_k.

Intuitively, the walkabout strength of a variable indicates the index of the weakest constraint used to compute the value of the variable. In the `for` loop from line 1, the algorithm determines w_j so that w_j indicates the index of the weakest constraint used to compute the value of y_j with $L\boldsymbol{y} = \boldsymbol{c}$ (Note that each pair of $\boldsymbol{l}_j$ and c_j corresponds to a constraint in the original HLS). Then, in the loop from line 5, it alters w_j based on the forms P, Q, and U of transformation matrices.

As noted, the algorithm for adding a constraint to an HLS predicts a constraint to revoke using walkabout strengths. To illustrate the prediction technique, assume that the constraint is inserted between the $(i-1)$-th and i-th constraints of the original HLS, and that by omitting non-zero coefficients, it is represented as

$$a_{j_1}x_{j_1} + a_{j_2}x_{j_2} + \cdots + a_{j_l}x_{j_l} = c$$

where $a_{j_{l'}} \neq 0$ for each l', and also let

$$k = \min(w_{j_1}, w_{j_2}, \ldots, w_{j_l}).$$

Then the prediction is either of the following:

1. If k is smaller than i, there is no constraint to revoke.
2. Otherwise, the k-th constraint of the original HLS is the candidate to revoke.

Clearly, the predicted constraint is the weakest constraint used to calculate the value of variables constrained by the added one. We can summarize the points of this prediction technique as follows:

- In case 1 of the prediction, it is always correct. In this case, the added constraint should not be enforced, and therefore, the algorithm can immediately terminate without any operations for modifying triangular factorization.
- In case 2 of the prediction, the algorithm can usually confirm its correctness by applying transformation matrices to the coefficient vector of the added constraint. It means that, in most cases, we only need to incrementally modify the triangular factorization, which is much less time-consuming than the basic strategy.

If the algorithm cannot ensure the correctness of the prediction (the prediction may be a miss), it switches to the basic strategy to guarantee a correct solution. Although the basic strategy is expensive, this case is rare as far as we experienced.

Removing a Constraint from an HLS

When we remove a constraint from an HLS whose triangular factorization is already computed, we have either of the following two cases:

- The constraint to remove is enforced. In this case, we delete the corresponding row from the matrix B, and add a coefficient vector of another weaker unenforced constraint instead to the bottom. Then we incrementally obtain the new triangular factorization.
- The constraint to remove is unenforced. Then we do not need to modify the current factorization.

The algorithm for removing a constraint from an HLS can be outlined as follows:

1. If the constraint to remove is unenforced, the algorithm terminates without revising the triangular factorization. Otherwise, remove the

corresponding row from the matrix B, and partially obtain the triangular factorization of the remaining rows incrementally.[4]

2. Select the strongest constraint among unenforced ones weaker than the removed one. Then apply the transformation matrices to the coefficient vector of the selected constraint. If the n-th entry of the resulting vector is not zero, enforce the constraint, determining the last transformation matrix for changing the n-th entry into 1. Otherwise, repeat this step by choosing the next strongest unenforced constraint.

At the step 2, if the n-th entry of the vector is not zero, the corresponding previously unenforced constraint is linearly independent of constraints used for the factorization. It means that the constraint is hierarchically independent since we search for such a constraint in the strength order.

Remarks

We make minor remarks on improving the performance of the algorithm.

Re-factorization. Re-factorization is also useful for improving the hit ratio of prediction of constraints to revoke in the incremental algorithm for adding a constraint. Constraints previously deleted from the matrix B may have left connections of variables that do not currently exist. Because of such connections, walkabout strengths of variables sometimes indicate indices of unrelated constraints that are weaker than the one to be really revoked.[5] Re-factorization cleans up such wrong connections, and enhances the possibility of correct prediction.

Filtering constraints to try to enforce. When the incremental algorithm removes a constraint, it tries to enforce unenforced weaker constraints one by one in the order from the strongest to the weakest. If there are many such unsatisfied constraints, this repetitive process may degrade the performance of constraint satisfaction.

[4] At this time, the matrix is $(n-1) \times n$ because the new row is not added yet. However, we can apply the technique for modifying triangular factorizations to such a matrix. In this case, we cannot determine the last transformation matrix of the form U, which is to transform the row to be enforced.

[5] In this case, the algorithm applies the basic strategy for incremental factorization. Although such wrong connections may degrade the performance, they never cause incorrect solutions.

We can reduce constraints to try using walkabout strengths. After deleting the row corresponding to the removed constraint, we calculate walkabout strengths by assigning the special value $m+1$ to w_n at line 3 of the algorithm of walkabout strengths. When the algorithm finishes, some variables have walkabout strengths $m+1$. We need to try only constraints referring such variables, because a constraint not referring such variables has no relation to the removed constraint.

6.3 Implementation

Based on the HiRise algorithm, we implemented a constraint solver in C++. As described, HiRise works as a solver for constraint hierarchies consisting of linear equality constraints. To realize interactive GUIs, it provides two-phase constraint satisfaction like the *DETAIL* constraint solver.

We implemented matrices of forms U and L using lists, not arrays, since they tend to be sparse in actual applications. For applications generating large constraint hierarchies, the list implementation can be expected to contribute to less time and less space. Also, we defined P and Q as simple fixed-size structures since we can express them with indices of columns to swap or shift.

In practical applications, variables are dynamically added to or removed from an HLS; that is, $\boldsymbol{x}$ is lengthened or shortened. Using the HiRise constraint solver, programmers can freely create and destroy variables, which are implemented as ordinary objects in C++. We supported addition and removal of variables in the following ways:

- When a constraint is added, the solver examines whether each variable referred by the constraint is already registered. If a variable is new to the solver, it records the variable in its internal variable list, which can be regarded as $\boldsymbol{x}$. Although $\boldsymbol{x}$ is lengthened, we do not need to modify the current matrices of forms P, Q, U, and L, since we implemented them not as arrays but as lists or simple structures using indices.

- When an existing variable is destroyed (which is reported by the destructor of the variable), the solver simply marks 'null' at its position in its variable list. We do not shorten the variable list, not only because such a task is time-consuming, but also because there may be wrong connections among variables (remind that revoked constraints leave their 'phantom' in the incremental algorithm). In the incremental algorithm, such null variables hardly degrade the performance although

they may waste a little memory.

Since the HiRise constraint solver is implemented in C++, it can be easily incorporated into various platforms, as opposed to the *DETAIL* constraint solver, written in Objective C.

For example, Figure 6.1 is a screen snapshot of a sample GUI application using HiRise, which runs on Windows 95 or NT 4.0. It allows a user to edit general graphs by adding nodes, fixing positions of nodes, and fixing directions of edges. In this application, there are two kinds of nodes: nodes of one kind, which are supplied by the application, are located in a circle when the application is invoked; nodes of the other kind, which are added by the user, are positioned at the centers of gravity of adjacent nodes. Therefore, the positions of the latter nodes are determined by the former nodes, and thus the application realizes automatic graph drawing.

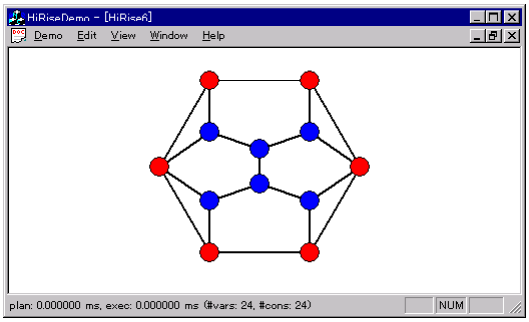

Figure 6.1: A sample GUI application using HiRise.

6.4 Performance Evaluation

This section evaluates the performance of the HiRise constraint solver.

6.4.1 Time Complexity

First, we provide the time complexity of the HiRise algorithm for satisfying HLSs. For simplicity, we assume that the numbers n and m of variables and constraints are large enough, and also that the count of incrementally added or removed constraints is much smaller than n and m. Then primary parts of the algorithm have the following time complexities:

- The time complexity of obtaining a triangular factorization from scratch is $O(mn^2)$.
- The time complexity of incrementally modifying a triangular factorization is $O(n^2)$.
- The time complexity of calculating variable values with a triangular factorization is $O(n^2)$.
- The time complexity of computing walkabout strengths of variables is $O(n^2)$.

Based on these complexities, we can estimate the time complexity of each operation as follows:

- The planning phase for non-incremental constraint satisfaction requires $O(mn^2)$ time.
- The planning phase for incrementally adding a constraint costs $O(n^2)$ time if a constraint to revoke is successfully detected with walkabout strengths (that is, only the enhanced strategy is performed). Otherwise (that is, the basic strategy is necessary), it takes $O(mn^2)$ time.
- The planning phase for incrementally removing a constraint needs $O(m'n^2)$ time, where m' indicates the number of unenforced constraints to try to satisfy instead of the removed one.
- The execution phase takes $O(n^2)$ time.

6.4.2 Experimental Results

We actually measured the performance of the HiRise constraint solver with a few experiments. With this measurement, we evaluated the effect of walkabout strengths on improving the efficiency of constraint satisfaction. For this purpose, we compared the performance of the complete version of HiRise with that of a version of HiRise that we disabled in compile time from the enhanced strategy for incremental satisfaction of preferential constraints.

For these experiments, we compiled the HiRise constraint solver using Microsoft Visual C++ 5.0 with the /O2 option, and executed test applications on a PC/AT compatible computer with a 266 MHz Intel Pentium II processor that runs the Microsoft Windows NT 4.0 operating system.

Binary Trees

The first experiment used an interactive application that allows us to edit a binary tree by dragging a node in the tree, adding a node to the tree, and removing a subtree from the tree. Figure 6.2 is a snapshot of the application, which provides a binary tree whose height is 4.

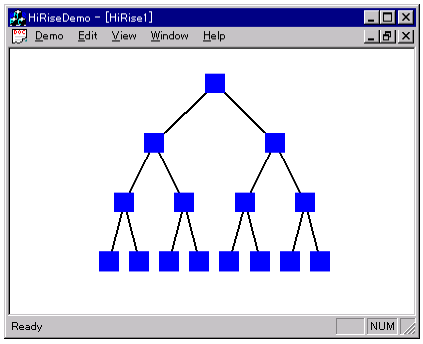

Figure 6.2: The application for editing binary trees.

Basically, the application layouts binary trees so that distances between their neighboring leaves become equal. Their definition of constraints are recursive, constructing a tree from its subtrees. In other words, it initially generates leaf nodes as base subtrees, and then recursively produces a larger subtree using two subtrees.

Each leaf node has four fields `x`, `y`, `left`, and `right`: fields `x` and `y` indicate the position of the node; fields `left` and `right` express the left and right bounds of the bounding box of the subtree whose top is the designated node.

For each leaf node `node`, we define the following constraints:

$$\texttt{node.left} + \texttt{x\_unit} = \texttt{node.right} \tag{6.4}$$

$$\texttt{node.x} = (\texttt{node.left} + \texttt{node.right})/2. \tag{6.5}$$

Intuitively, (6.4) means that the width of the 'territory' of `node` is `x_unit`, which is a global variable. This constraint is for locating leaf nodes at equal intervals. Also, (6.5) centers `node` in the territory.

Each intermediate node has the additional two fields, `left_node` and `right_node`, pointing to its children. For each intermediate node `node`, we

define the following constraints:

(6.6) `node.left_child.right` = `node.right_child.left`

(6.7) `node.left` = `node.left_child.left`

(6.8) `node.right` = `node.right_child.right`

(6.9) `node.x` = (`node.left_child.x` + `node.right_child.x`)/2

(6.10) `node.y` + `y_unit` = `node.left_child.y`

(6.11) `node.y` + `y_unit` = `node.right_child.y`.

Intuitively, (6.6) places the children of `node` side by side. (6.7) and (6.8) determine the width of the bounding box of `node` based on those of its children. (6.9) puts `node` at the center of its children. (6.10) and (6.11) put the children `y-unit` under `node`.

Layouting binary trees based on this definition requires simultaneous satisfaction of constraints. Therefore, classical local propagation constraint solvers such as DeltaBlue cannot treat this layout. Also, although more sophisticated local propagation solvers supporting cycles or constraint cells, e.g. SkyBlue and *DETAIL*, handle the layout, it is difficult to efficiently solve the constraint hierarchy for the layout; for a single binary tree, there exists a large constraint cell that must vary as constraints are added or removed because of the user's manipulation. Thus the layout is a hard problem for most local propagation solvers.

Numbers of variables and constraints can be estimated as follows: consider a binary tree whose height is k; then the number of its nodes is $2^k - 1$, the number of its variables is $2^{k+2} - 4$, and the number of its constraints is $2^{k+2} - 6$. The actual application yields a slightly larger number of constraints by introducing weak constraints to dictate the behavior of binary trees.

We performed two experiments: for the first experiment, we defined all the constraints described above as required ones; for the second experiment, we modified (6.4), (6.10), and (6.11) into strong preferential constraints. Consequently, the ratios of required constraints in the first and second experiments are approximately one hundred percent and sixty percent respectively.

Table 6.1 shows the results of the first experiment. In this table, 'basic' indicates the version of HiRise that we disabled from the enhanced strategy for incremental satisfaction of preferential constraints, and 'complete' points to the normal version of HiRise, which is capable of both of the basic and enhanced strategies. Also, 'initial,' 'drag,' 'stay,' 'line,' 'add,' and 'remove'

respectively denote the user's manipulations for obtaining an initial layout, dragging a node, staying a node, constraining a node to be on some straight line, adding a new node, and removing an existing node.

height		8		9		10	
# all constraints		1022		2046		4094	
# required constraints		1018		2042		4090	
phase		plan	exec	plan	exec	plan	exec
basic	initial	270	<10	1172	20	6809	81
	drag	<10	10	10	20	30	90
	stay	<10	10	10	20	30	80
	line	<10	10	10	20	20	80
	add	10	10	50	20	161	90
	remove	10	10	50	20	170	90
complete	initial	261	10	1161	31	6189	80
	drag	20	10	80	20	310	80
	stay	20	<10	10	20	30	80
	line	10	10	50	30	220	91
	add	60	10	230	20	861	91
	remove	40	<10	120	20	471	90

Table 6.1: Times in milliseconds to edit binary trees defined with required constraints.

The results show that both of the versions of HiRise cost similar times to obtain initial layouts. This is because obtaining initial layouts requires non-incremental constraint satisfaction, for which they work almost equally. For the other editing operations, both of the versions took much less times than obtaining initial layouts since they performed incremental constraint satisfaction. These results suggest that HiRise enables real-time interaction even for thousands of constraints. By contrast to the non-incremental case, incremental constraint satisfaction of the complete version was a few times to tens of times slower than that of the basic version. We can understand that maintaining walkabout strengths became an overhead in the complete version because most constraints were required in this experiment.

Next, Table 6.2 gives the results of the second experiment. As noted, this is a case that the percentage of required constraints is about sixty. In this experiment, the basic version of HiRise exhibited a remarkable slowdown, which were hardly acceptable to real-time interaction. By contrast, the

complete version performed stably.[6]

height		8		9		10	
# all constraints		1022		2046		4094	
# required constraints		636		1276		2556	
phase		plan	exec	plan	exec	plan	exec
basic	initial	230	<10	922	10	4647	10
	drag	130	<10	511	<10	2944	<10
	stay	120	<10	520	<10	2924	<10
	line	120	<10	511	<10	2884	10
	add	130	<10	541	<10	3015	<10
	remove	150	<10	591	<10	3235	<10
complete	initial	231	<10	912	<10	4226	11
	drag	10	<10	10	<10	30	<10
	stay	10	<10	10	<10	30	<10
	line	<10	<10	10	<10	30	<10
	add	20	<10	30	<10	80	10
	remove	10	<10	20	<10	60	10

Table 6.2: Times in milliseconds to edit binary trees defined with required and preferential constraints.

General Trees

Next, we measured the performance of HiRise using general trees. The application utilized was developed by extending the previous application for editing binary trees so that a single node could have an arbitrary number of children as depicted in Figure 6.3.

This application generates random trees that are not balanced. Specifically, when a user invokes the application, the user is prompted to input the height of a binary tree to generate, together with the maximum number of children that a single intermediate node tree can have. Then the application will automatically generate a random tree according to the user's input. In this experiment, we used trees whose height is 9 and the maximum number of whose children is 4.[7]

[6] Actually, it became faster than the first experiment. We can understand that it was an accidental result due to sparseness of HLSs. Such cases would decrease if we treated sparseness better by analyzing HLSs with, e.g., ordering.

[7] The user can also specify a seed of random numbers. Using this feature, we obtained

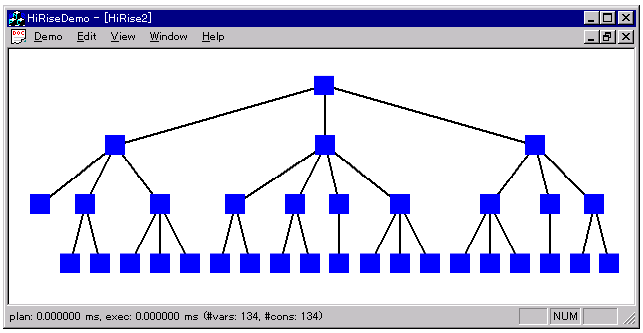

Figure 6.3: The application for editing general trees.

In the same way as the previous experiment on binary trees, we examined two cases that the ratios of required constraints were approximately one hundred percent and sixty percent. As Table 6.3 shows, the complete version of the HiRise constraint solver exhibited stable performance throughout this experiment.

# all constraints		2882		2882	
# required constraints		2878		1730	
phase		plan	exec	plan	exec
basic	initial	2314	40	2023	10
	drag	10	40	1242	10
	stay	10	50	1212	10
	line	10	50	1211	10
	add	80	50	1241	10
	remove	90	50	1312	<10
complete	initial	2484	40	2023	<10
	drag	180	50	30	<10
	stay	180	51	20	<10
	line	120	40	20	<10
	add	420	41	50	10
	remove	201	50	30	10

Table 6.3: Times in milliseconds to edit general trees.

the same shape of trees throughout this experiment.

Koch's Curve

Finally, we investigated the performance of HiRise using Koch's curve, which is known as a fractal figure. The application that we constructed displays a polyline that is an approximation of Koch's curve at a given level as illustrated in Figure 6.4, and allows its user to resize the polyline by dragging one of its vertices, to move it by dragging its top vertex, and to give an additional detail locally to one of its edges as depicted in Figure 6.5.

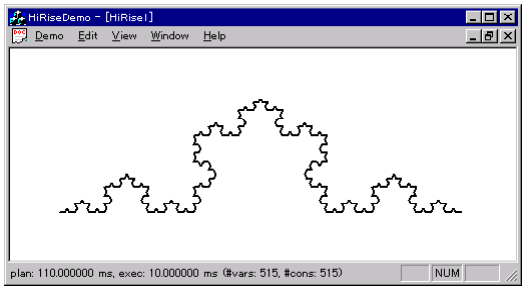

Figure 6.4: The application for manipulating Koch's curve.

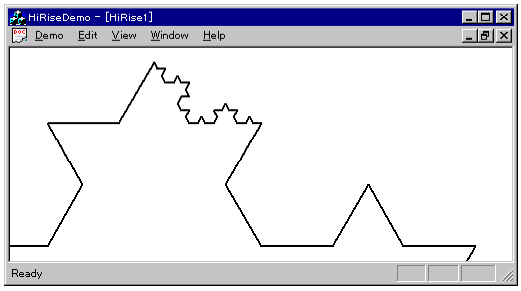

Figure 6.5: Adding details locally to the approximation of Koch's curve.

In this experiment, we used an approximation of Koch's curve at level 5. Initially, the application generates 2051 constraints.

This experiment showed that the complete version of HiRise usually performed stably in incremental constraint satisfaction. However, in moving the figure by dragging its top vertex, the complete version required a little longer time for planning than the basic version. It was because there may exist cases that the enhanced strategy for incremental satisfaction of

preferential constraints is not applicable; in such cases, after failing in the enhanced strategy, HiRise must apply the basic strategy. However, such cases are quite rare, and also, even if such a case occurs, the unsuccessful enhanced strategy will take a much shorter time than the basic strategy to follow. Thus it is not disadvantageous to perform the enhanced strategy before the basic one.

# all constraints		2051		2051	
# required constraints		2048		1365	
phase		plan	exec	plan	exec
basic	initial	1592	20	2093	40
	resize	10	20	1301	50
	move	20	20	1332	40
	add	30	30	1282	40
complete	initial	1593	20	2083	50
	resize	120	20	201	40
	move	60	30	1372	40
	add	261	20	441	40

Table 6.4: Times in milliseconds to manipulate Koch's curve.

6.5 Discussion

This section provides minor discussions on the HiRise constraint solver.

6.5.1 Techniques for Sparse Matrices

Currently, the HiRise constraint solver does not incorporate known optimization techniques for sparse matrices. However, it is possible to apply certain preprocessing techniques for sparse matrices, e.g. ordering, to a given set of required constraints, and they are expected to be effective for speedup of constraint satisfaction. To the contrary, it is difficult to apply such techniques to preferential constraints, because it is not possible to simply alter their order.

Another approach to sparse matrices might use iterative methods, which satisfy non-hierarchical linear systems by repeatedly finding better approximate solutions, and which are widely adopted to solve sparse systems. To introduce iterative methods is quite difficult in our current framework of HLSs, and we must need additional basic studies on HLSs.

6.5.2 Least-Squares Method

As we described in Subsection 5.5.2, we can incorporate into HLSs a hybrid comparator capable of the least-squares method. Also, we can accommodate the HiRise algorithm to the least-squares method.

The basic idea is to use walkabout strengths to find constraints to relax. When we add a preferential constraint, we can find, with walkabout strengths, an existing constraint to revoke. If the added one and the found one are constraints to be relaxed together, we can replace them with another constraint that indicates the relaxed form of them.

This technique depends on the fact that the over-constrained set of linear equations to be solved with the least-squares method, $A\boldsymbol{x} = \boldsymbol{c}$, can be represented as a set of ordinary linear equations as follows [2]:

$$A^T A\boldsymbol{x} = A^T \boldsymbol{c}.$$

Chapter 7

Related Work

This chapter describes previous researches on constraints, especially from the viewpoints of treatment and satisfaction of over-constrained systems.

7.1 Research Areas on Constraints

Since constraints are general tools for problem solving, many researches in various areas introduce constraints for their own purposes. The main areas are as follows:

Artificial intelligence employs constraints for a variety of problems such as scheduling, reasoning, belief maintenance, and machine vision [53, 78, 85]. Most of such problems are generally formalized as *constraint satisfaction problems* (CSPs), which handle finite domains, and have been extensively studied [53, 85].

Logic programming adopts constraints to expand problem domains by replacing the traditional unification mechanism with more general constraint satisfaction techniques. Such an extension is called constraint logic programming [17]. Representative of constraint logic programming languages are Prolog III [18] and CLP(R) [48].

Graphical user interfaces (GUIs) utilize constraints to manage internal data and visual objects [68]. This approach can be back to the Sketchpad system [79], but the current stream seems to have its source at an object-oriented system called ThingLab [3, 6].

Another research area is, for example, imperative programming [20, 21, 35, 54, 55, 56, 57] and databases.

Usually, distinct areas take different approaches to constraints. Also, they employ different algorithms for constraint satisfaction. In the rest of this chapter, we mainly review researches related to GUIs.

7.2 Ordinary Constraint Systems

In this section, we focus on ordinary constraint systems, which do not handle over-constrained situations (theoretically, they produce empty solution sets for such situations).

For ordinary systems, many numerical methods for satisfying systems of algebraic constraints, such as linear equations, linear inequalities, and quadratic equations, can be used [33, 45, 46, 67]. The Oak system realizes real-time constraint satisfaction by employing linear approximation for quadratic constraints [84]. Snap-together mathematics is a technique for approximation of constraint systems based on dynamic models and solves them efficiently [26, 27, 28].

Local propagation has often been applied to ordinary constraint systems. Sketchpad is the first GUI system that adopts a local propagation technique called 'one-pass method' (it also uses numerical methods on failure of the one-pass method) [79]. Most systems employ one-way constraints, whose output variables are predetermined by programmers [42, 63, 64, 65, 66, 81]. By contrast, Fabrik handles two-way constraints, each of which has two potential output [47], and ThingLab [3, 6] and CONSTRAINTS [78] exploit multi-way constraints, where each variable can be an output.

In the context of local propagation for ordinary systems, researchers have been investigated various topics, for example, incremental satisfaction of constraint systems [43, 88], treatment of pointer variables [41, 87], compilation of constraint systems [89], support of computer-supported cooperative work [34], and 'light' implementation of constraints [44].

Another technique for solving ordinary systems is transformation. GITS uses a simple technique that replaces constraints using a pre-defined table [69]. The Magritte graphic editor utilizes algebraic transformation [29]. Systems employing term rewriting are also proposed [35, 89].

Depart from GUIs, techniques for solving constraint systems over finite domains have been extensively studied in the context of CSPs [85]. Traditional approaches are search techniques such as backtracking and forward checking [85]. Recently, problem reduction techniques using 'arc consistency' are often explored [1, 32, 53, 85].

7.3 Least-Squares Problems

One of the simplest approaches for treating over-constrained situations is the least-squares method [2]. In this approach, conflicting constraints in over-constrained systems are relaxed by minimizing sums of squares of their errors.

The ThingLab system, which normally uses local propagation, performs a hill-climbing least-squares technique called relaxation in case local propagation is not applicable [3, 6]. The TRIP systems divide constraint systems consisting of linear equations into two levels, and applies the least-squares method to the weaker level [52, 59, 62, 82, 83].

7.4 Constraint Hierarchies

This section presents previous work on constraint hierarchies from the viewpoints of theories and algorithms.

7.4.1 Theories

Borning et al. formulated constraint hierarchies [7, 9, 11, 90, 91, 92, 93], and studied their properties [9, 91]. They also integrated constraint hierarchies with logic programming as hierarchical constraint logic programming (HCLP), and explored theoretical properties of HCLP [11, 90, 91, 92, 93].

Jampel constructed a certain HCLP instance that separates the HCLP scheme into compositional and non-compositional parts [50]. The method is expected to improve the efficiency of interpreters and compilers since the compositional part is efficiently implementable.

Wolf formalized ordered constraint hierarchies by defining a hierarchy comparator that makes constraints totally ordered in each level, and also provided a method for transforming ordered constraint hierarchies into ordinary constraint systems [94].

7.4.2 Algorithms

This subsection presents past researches on satisfaction of constraint hierarchies by categorizing them into the refining method, the optimization approach, local propagation, and others.

The Refining Method

Borning's early system incorporating constraint hierarchies applied the relaxation technique to each level of the hierarchies, and obtained least-squares-better solutions [7].

The Orange algorithm finds one or all solutions of hierarchies of linear equality and inequality constraints by performing the simplex method in each level [22].

DeltaStar is a general algorithm developed for an HCLP interpreter. It transforms a constraint hierarchy solved with weighted-sum-metric-better, worst-case-better, or locally-metric-better into a series of linear programming problems by successively solving each level from the strongest to the weakest [90].

Indigo is an algorithm that finds locally-metric-better solutions of acyclic (i.e. non-simultaneous) hierarchies of inequality constraints [5]. It is unique in that it performs interval propagation to handle inequalities instead of a numerical technique, and is efficient enough to realize GUIs.

The Optimization Approach

Cassowary and QOCA are constraint solvers for GUIs that maintain hierarchies of linear equality and inequality constraints using optimization techniques [12]. Cassowary obtains locally-error-better solutions by using the simplex method for linear programming, and QOCA finds least-squares-better solutions by using the active set method for quadratic programming.

Local Propagation

Blue is an algorithm that obtains one locally-predicate-better solution of a constraint hierarchy consisting of multi-way constraints [58], and DeltaBlue is an incremental version of Blue [22, 58, 76]. Both of them embody a problem that they cannot solve hierarchies involving cycles. Suzuki et al. modified DeltaBlue in three ways to speed up its planning phase [80].

SkyBlue is an incremental algorithm for solving hierarchies of multi-way constraints [70, 71, 72, 73, 74, 75]. It addresses simultaneous constraints using pluggable cycle solvers, and handles constraints with multi-output methods.

QuickPlan is an algorithm for incrementally resolving hierarchies of multi-way constraints [86]. It can solve constraint hierarchies if they have at least one acyclic solution graph.

Houria [14] and Houria II [15] are algorithms that solve constraint hierarchies using a certain kind of global hierarchy comparators.

Others

Techniques for compiling constraint hierarchies into imperative procedures are proposed [19, 30, 58]. They them limit constraint hierarchies to being statically defined, that is, they cannot dynamically change hierarchies after compilation.

UltraViolet is a hybrid algorithm that divides constraint hierarchies into subgraphs based on graph topology and constraint types and then invokes appropriate subsolvers [8, 10].

Menezes's incremental hierarchical constraint solver satisfies constraint hierarchies over finite domains using a search technique [60].

7.5 Other Over-Constrained Systems

Hattori proposed executable constraints [31]. Each executable constraint is a macro defined by an end user of a GUI system, and the system computes solutions by discarding executable constraints that cause conflicts.

In the area of artificial intelligence, various approaches have been provided to handle over-constrained CSPs [49]. Freuder and Wallace proposed partial constraint satisfaction for handling constraint satisfaction problems that are impossible or impractical to solve, and also presented algorithms searching for approximate solutions by 'weakening' CSPs over finite domains [23, 24]. Jampel provided a formal method for transformations between constraint hierarchies and partial constraint satisfaction problems [51].

Chapter 8

Conclusion and Future Work

This chapter concludes the dissertation and provides future directions.

8.1 Conclusion

In this dissertation, we discussed properties and satisfaction of hierarchical constraint systems (HCSs) from the viewpoint of local propagation. Basically, local propagation is a method that gradually satisfies constraints by scheduling them beforehand. Also, it is usually associated with incremental constraint satisfaction.

Since classical local propagation algorithms solved constraints one by one, local propagation has been considered to be restricted to dataflow constraints. However, in this research, we conceived that the essence of local propagation is its unique strategy of scheduling constraints—it sometimes schedules weak constraints prior to stronger ones.

Based on this idea, we formalized generalized local propagation (GLP) by targeting one of the most popular formulations of HCSs, the theory of constraint hierarchies. With GLP, we revealed not only that we can introduce simultaneous constraint satisfaction into local propagation algorithms, but also that we may extend methods for satisfying constraint hierarchies from local satisfiers to an important class of global satisfiers via a concept that we call global semi-monotonicity.

Using the result of GLP, we contrived the *DETAIL* algorithm by extending conventional incremental local propagation algorithms for satisfying dataflow constraints. It is the first local propagation algorithm that simulates a global satisfier.

Conventional local propagation algorithms, including *DETAIL*, uniquely

satisfies something corresponding to a block in GLP at each step. However, GLP itself does not compel us to satisfy blocks uniquely. Thus we attempted to apply the background idea of GLP to numerical algorithms by switching the kind of constraints from dataflow ones to algebraic ones.

To show the idea concretely, we adopted linear equality constraints as algebraic constraints, and defined hierarchical linear systems (HLSs), which can be viewed as a specialization of constraint hierarchies in linear equality constraints. Then we successfully obtained constraint scheduling like GLP using a notion called hierarchical independence. Because of the algebraic properties of linear constraints, we could become more aggressive in constraint scheduling than GLP.

Employing the result of HLSs, we designed the HiRise algorithm, which solves HLSs numerically. We proved that local propagation contributes to efficiency by associating it with incremental constraint satisfaction.

In summary, we disclosed that local propagation is a general approach to satisfaction of HCSs with various constraints, not a special method limited to dataflow constraints.

8.2 Future Work

Finally, we describe our future work.

8.2.1 Enhancing the GLP Theory

The current GLP theory discusses sufficient conditions for guaranteeing that a solution generated by a single ordered partition is correct. However, we cannot always satisfy constraint hierarchies based only on the conditions. For example, the HiRise algorithm must sometimes do non-incremental constraint satisfaction because it cannot guarantee the correctness of the solution using walkabout strengths. Thus we need more powerful methods.

A possible direction is that we treat 'dynamic' update of ordered partitions as well as such a single 'static' ordered partition as is with the current GLP theory. For example, it may be useful to discuss the SkyBlue algorithm, which obtains solutions using backtracking.

8.2.2 More Powerful Constraint Solvers

Our primary goal is to develop powerful constraint solvers. Since *DETAIL* and HiRise are not our ultimate goals, we must pursue more powerful constraint solvers. For this goal, we try various approaches including, for exam-

ple, the optimization approach. However, we believe that the spirit of local propagation will also survive by integrating into other approaches.

Bibliography

[1] Bessière, C. and M.-O. Cordier, "Arc-Consistency and Arc-Consistency Again," in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, July 1993, pp. 108–113.

[2] Björck, Å., *Numerical Methods for Least Squares Problems.* Philadelphia: SIAM, 1996.

[3] Borning, A., "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4, Oct. 1981, pp. 353–387.

[4] Borning, A., "Problem with SkyBlue and Cycles." Posted to comp.constraints (also available at http://www.cs.washington.edu/research/constraints/skyblue-cycles.html), Mar. 1995.

[5] Borning, A., R. Anderson, and B. Freeman-Benson, "Indigo: A Local Propagation Algorithm for Inequality Constraints," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1996, pp. 129–136.

[6] Borning, A. and R. Duisberg, "Constraint-Based Tools for Building User Interfaces," *ACM Transactions on Graphics*, vol. 5, no. 4, Oct. 1986, pp. 345–374.

[7] Borning, A., R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf, "Constraint Hierarchies," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 1987, pp. 48–60.

[8] Borning, A. and B. Freeman-Benson, "UltraViolet: A Constraint Satisfaction Algorithm for Interactive Graphics," *CONSTRAINTS: An International Journal*, To appear.

[9] Borning, A., B. Freeman-Benson, and M. Wilson, "Constraint Hierarchies," *Lisp and Symbolic Computation*, vol. 5, no. 3, Sept. 1992, pp. 223–270.

[10] Borning, A. and B. N. Freeman-Benson, "The OTI Constraint Solver: A Constraint Library for Constructing Interactive Graphical User Interfaces," in *Principles and Practice of Constraint Programming—CP'95*, vol. 976 of *Lecture Notes in Computer Science*, Springer-Verlag, Oct. 1995, pp. 624–628.

[11] Borning, A., M. Maher, A. Martindale, and M. Wilson, "Constraint Hierarchies and Logic Programming," in *Proceedings of the Sixth International Conference on Logic Programming*, June 1989, pp. 147–164.

[12] Borning, A., K. Marriott, P. Stuckey, and Y. Xiao, "Solving Linear Arithmetic Constraints for User Interface Applications," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Oct. 1997, pp. 87–96.

[13] Boundy, J. A. and U. S. R. Murty, *Graph Theory with Applications.* Macmillan Press, 1976.

[14] Bouzoubaa, M., B. Neveu, and G. Hasle, "Houria: A Solver for Equational Constraints in a Hierarchical System," in *Proceedings of the Workshop on Over-Constrained Systems at CP'95*, Sept. 1995.

[15] Bouzoubaa, M., B. Neveu, and G. Hasle, "Houria II: A Solver for Hierarchical Constraint Systems," in *Proceedings of the Workshop on Constraints for Graphics and Visualization at CP'95*, Sept. 1995.

[16] Chvátal, V., *Linear Programming.* New York: Freeman, 1983.

[17] Cohen, J., "Constraint Logic Programming Languages," *Communications of the ACM*, vol. 33, no. 7, July 1990, pp. 52–68.

[18] Colmerauer, A., "An Introduction to Prolog III," *Communications of the ACM*, vol. 33, no. 7, July 1990, pp. 69–90.

[19] Freeman-Benson, B. N., "A Module Mechanism for Constraints in Smalltalk," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 1989, pp. 389–396.

[20] Freeman-Benson, B. N. and A. Borning, "The Design and Implementation of Kaleidoscope'90, A Constraint Imperative Programming Language," in *Proceedings of the IEEE Conference on Computer Languages*, Apr. 1992, pp. 174–180.

[21] Freeman-Benson, B. N. and A. Borning, "Integrating Constraints with an Object-Oriented Language," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, vol. 615 of *Lecture Notes in Computer Science*, Springer-Verlag, June/July 1992, pp. 268–286.

[22] Freeman-Benson, B. N., J. Maloney, and A. Borning, "An Incremental Constraint Solver," *Communications of the ACM*, vol. 33, no. 1, Jan. 1990, pp. 54–63.

[23] Freuder, E. C., "Partial Constraint Satisfaction," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Aug. 1989, pp. 278–283.

[24] Freuder, E. C. and R. J. Wallace, "Partial Constraint Satisfaction," *Artificial Intelligence*, vol. 58, 1992, pp. 21–70.

[25] Gangnet, M. and B. Rosenberg, "Constraint Programming and Graph Algorithms," in *Proceedings of Second International Symposium on Artificial Intelligence and Mathematics*, Jan. 1992.

[26] Gleicher, M., "A Graphical Toolkit Based on Differential Constraints," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1993, pp. 109–120.

[27] Gleicher, M. and A. Witkin, "Differential Manipulation," in *Proceedings of Graphics Interface '91*, June 1991, pp. 61–67.

[28] Gleicher, M. and A. Witkin, "Supporting Numerical Computations in Interactive Contexts," in *Graphics Interface '93*, May 1993, pp. 138–145.

[29] Gosling, J., "Algebraic Constraints," Tech. Rep. CMU-CS-83-132, Department of Computer Science, Carnegie-Mellon University, May 1983.

[30] Harvey, W., P. Stuckey, and A. Borning, "Compiling Constraint Solving using Projection," in *Proceedings of the Third International Conference on the Principles and Practice of Constraint Programming*, 1997.

[31] Hattori, T., "Integration of Macros and Constraints in Editors," in *Interactive Systems and Software IV (JSSST WISS'96)* (J. Tanaka, ed.), vol. 16 of *Lecture Notes in Software Science*, Kindai-Kagaku-Sha, Dec. 1996, pp. 41–49. In Japanese.

[32] Hentenryck, P. V., Y. Deville, and C.-M. Teng, "A Generic Arc-Consistency Algorithm and its Specializations," *Artificial Intelligence*, vol. 57, 1992, pp. 291–321.

[33] Heydon, A. and G. Nelson, "The Juno-2 Constraint-Based Drawing Editor," Research Report 131a, Digital Systems Research Center, Dec. 1994.

[34] Hill, R. D., "The Rendezvous Constraint Maintenance System," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1993, pp. 225–234.

[35] Horn, B., "Constraint Patterns As a Basis For Object Oriented Programming," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 1992, pp. 218–233.

[36] Hosobe, H., Matsuoka, and A. Yonezawa, "Efficient Satisfaction of Constraint Hierarchies with Inequalities," in *Interactive Systems and Software III (JSSST WISS'95)* (J. Tanaka, ed.), vol. 12 of *Lecture Notes in Software Science*, Kindai-Kagaku-Sha, Dec. 1995, pp. 123–132. In Japanese.

[37] Hosobe, H., Matsuoka, and A. Yonezawa, "Efficient Satisfaction of Constraint Hierarchies Using Hierarchical Linear Systems," in *Interactive Systems and Software V (JSSST WISS'97)* (R. Onai, ed.), vol. 18 of *Lecture Notes in Software Science*, Kindai-Kagaku-Sha, Dec. 1997, pp. 129–134. In Japanese.

[38] Hosobe, H., S. Matsuoka, and A. Yonezawa, "Generalized Local Propagation: A Framework for Solving Constraint Hierarchies," in *Principles and Practice of Constraint Programming—CP'96* (E. C. Freuder, ed.), vol. 1118 of *Lecture Notes in Computer Science*, Springer-Verlag, Aug. 1996, pp. 237–251.

[39] Hosobe, H., K. Miyashita, S. Takahashi, S. Matsuoka, and A. Yonezawa, "Locally Simultaneous Constraint Satisfaction," in *Interactive Systems and Software I (JSSST WISS'93)* (A. Takeuchi, ed.), vol. 7 of *Lecture*

Notes in Software Science, Kindai-Kagaku-Sha, Sept. 1994, pp. 49–56. In Japanese.

[40] Hosobe, H., K. Miyashita, S. Takahashi, S. Matsuoka, and A. Yonezawa, "Locally Simultaneous Constraint Satisfaction," in *Principles and Practice of Constraint Programming—PPCP'94* (A. Borning, ed.), vol. 874 of *Lecture Notes in Computer Science*, Springer-Verlag, Oct. 1994, pp. 51–62.

[41] Hudosn, S. E., "A System for Efficient and Flexible One-Way Constraint Evaluation in C++." 1993.

[42] Hudson, S. E., "Graphical Specification of Flexible User Interface Displays," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1989, pp. 105–114.

[43] Hudson, S. E., "Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 3, July 1991, pp. 315–341.

[44] Hudson, S. E. and I. Smith, "Ultra-Lightweight Constraints," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1996, pp. 147–155.

[45] Igarashi, T., S. Matsuoka, S. Kawachiya, and H. Tanaka, "In Search for an Ideal Computer-Assisted Drawing System," in *Proceedings of the Sixth IFIP Conference on Human-Computer Interaction (INTERACT'97)*, July 1997, pp. 104–111.

[46] Igarashi, T., S. Matsuoka, S. Kawachiya, and H. Tanaka, "Interactive Beautification: A Technique for Rapid Geometric Design," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Oct. 1997, pp. 105–114.

[47] Ingalls, D., S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle, "Fabrik A Visual Programming Environment," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Sept. 1988, pp. 176–190.

[48] Jaffar, J., S. Michaylov, P. J. Stuckey, and R. H. C. Yap, "The CLP(R) Language and System," *ACM Transactions on Programming Languages and Systems*, vol. 14, no. 3, 1992, pp. 339–395.

[49] Jampel, M., "A Brief Overview of Over-Constrained Systems," in *Over-Constrained Systems*, vol. 1106 of *Lecture Notes in Computer Science*, Springer-Verlag, Aug. 1996, pp. 1–22.

[50] Jampel, M., "A Compositional Theory of Constraint Hierarchies (Operational Semantics)," in *Over-Constrained Systems*, vol. 1106 of *Lecture Notes in Computer Science*, Springer-Verlag, Aug. 1996, pp. 189–206.

[51] Jampel, M., J.-M. Jacquet, D. Gilbert, and S. Hunt, "Transformations between HCLP and PCSP," in *Principles and Practice of Constraint Programming—CP'96* (E. C. Freuder, ed.), vol. 1118 of *Lecture Notes in Computer Science*, Springer-Verlag, Aug. 1996, pp. 252–266.

[52] Kamada, T. and S. Kawai, "A General Framework for Visualizing Abstract Objects and Relations," *ACM Transactions on Graphics*, vol. 10, no. 1, Jan. 1991, pp. 1–39.

[53] Kumar, V., "Algorithms for Constraint-Satisfaction Problems: A Survey," *AI Magazine*, Spring 1992, pp. 32–44.

[54] Lopez, G., "The Design and Implementation of Keleidoscope, A Constraint Imperative Language (Ph.D. Dissertation)," Tech. Rep. 97-04-08, Dept. of Computer Science and Engineering, University of Washington, Apr. 1997.

[55] Lopez, G., B. Freeman-Benson, and A. Borning, "Kaleidoscope: A Constraint Imperative Programming Language," Tech. Rep. 93-09-04, Dept. of Computer Science and Engineering, University of Washington, Sept. 1993.

[56] Lopez, G., B. Freeman-Benson, and A. Borning, "Constraints and Object Identity," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, no. 821 in Lecture Notes in Computer Science, pp. 260–279, Springer-Verlag, July 1994.

[57] Lopez, G., B. Freeman-Benson, and A. Borning, "Implementing Constraint Imperative Programming Languages: the Kaleidoscope'93 Virtual Machine," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 1994, pp. 259–271.

[58] Maloney, J. H., A. Borning, and B. N. Freeman-Benson, "Constraint Technology for User-Interface Construction in ThingLab II," in *Proceed-*

ings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Oct. 1989, pp. 381–388.

[59] Matsuoka, S., S. Takahashi, T. Kamada, and A. Yonezawa, "A General Framework for Bi-Directional Translation between Abstract and Pictorial Data," *ACM Transactions on Information Systems*, vol. 10, no. 4, Oct. 1992, pp. 408–437.

[60] Menezes, F., P. Barahona, and P. Codognet, "An Incremental Hierarchical Constraint Solver," in *Proceedings of the First Workshop on Principles and Practice of Constraint Programming (PPCP'93)* (Saraswat and van Hentenryck, eds.), MIT Press, 1994.

[61] Miyashita, K., S. Matsuoka, S. Takahashi, and A. Yonezawa, "Interactive Generation of Graphical User Interfaces by Multiple Visual Examples," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1994, pp. 85–94.

[62] Miyashita, K., S. Matsuoka, S. Takahashi, A. Yonezawa, and T. Kamada, "Declarative Programming of Graphical Interfaces by Visual Examples," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1992, pp. 107–116.

[63] Myers, B. A., *Creating User Interfaces by Demonstration.* San Diego: Academic Press, 1988.

[64] Myers, B. A., "Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 2, Apr. 1990, pp. 143–177.

[65] Myers, B. A., D. A. Giuse, R. B. Dannenberg, B. Vander Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *IEEE Computer*, vol. 23, no. 11, Nov. 1990, pp. 71–85.

[66] Myers, B. A., D. A. Giuse, and B. Vander Zanden, "Declarative Programming in a Prototype-Instance System: Object-Oriented Programming without Writing Methods," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 1992, pp. 184–200.

[67] Nelson, G., "Juno: a Constraint-based Graphics System," *Computer Graphics (SIGGRAPH'85)*, vol. 19, no. 3, July 1985, pp. 235–243.

[68] Olsen, Jr., D. R., *User Interface Management Systems: Models and Algorithms*. San Mateo, California: Morgan Kaufmann Publishers, 1992.

[69] Olsen Jr., D. R. and K. Allan, "Creating Interactive Techniques by Symbolically Solving Geometric Constraints," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Oct. 1990, pp. 102–107.

[70] Sannella, M., "The SkyBlue Constraint Solver," Tech. Rep. 92-07-02, Dept. of Computer Science and Engineering, University of Washington, Feb. 1993.

[71] Sannella, M., "Analyzing and Debugging Hierarchies of Multi-way Local Propagation Constraints," in *Principles and Practice of Constraint Programming—PPCP'94* (A. Borning, ed.), vol. 874 of *Lecture Notes in Computer Science*, Springer-Verlag, Oct. 1994, pp. 63–77.

[72] Sannella, M., "Constraint Satisfaction and Debugging for Interactive User Interfaces," Tech. Rep. 94-09-10, Dept. of Computer Science and Engineering, University of Washington, Sept. 1994.

[73] Sannella, M., "SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1994, pp. 137–146.

[74] Sannella, M., "The SkyBlue Constraint Solver and its Applications," in *Proceedings of the First Workshop on Principles and Practice of Constraint Programming (PPCP'93)* (Saraswat and van Hentenryck, eds.), MIT Press, 1994.

[75] Sannella, M. and A. Borning, "Multi-Garnet: Integrating Multi-Way Constraints with Garnet," Tech. Rep. 92-07-01, Dept. of Computer Science and Engineering, University of Washington, July 1992.

[76] Sannella, M., B. Freeman-Benson, J. Maloney, and A. Borning, "Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm," Tech. Rep. 92-07-05, Dept. of Computer Science and Engineering, University of Washington, July 1992.

[77] Satoh, K. and A. Aiba, "Computing Soft Constraints by Hierarchical Constraint Logic Programming," Tech. Rep. TR-610, ICOT, Japan, Jan. 1991.

[78] Sussman, G. J. and G. L. Steele Jr., "CONSTRAINTS—A Language for Expressing Almost Hierarchical Descriptions," *Artificial Intelligence*, vol. 14, 1980, pp. 1–39.

[79] Sutherland, I. E., "Sketchpad: A Man-Machine Graphical Communication System," in *Proceedings of the AFIPS Spring Joint Conference*, vol. 23, 1963, pp. 329–346.

[80] Suzuki, T., N. Kakinuma, and T. Tokuda, "An Experimental Comparison of Three Modified DeltaBlue Algorithms," in *Principles and Practice of Constraint Programming—CP'96* (E. C. Freuder, ed.), vol. 1118 of *Lecture Notes in Computer Science*, Springer-Verlag, Aug. 1996, pp. 425–435.

[81] Szekely, P. A. and G. A. Myers, "A User Interface Toolkit Based on Graphical Objects and Constraints," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Sept. 1988, pp. 36–45.

[82] Takahashi, S., S. Matsuoka, A. Yonezawa, and T. Kamada, "A General Framework for Bi-Directional Translation between Abstract and Pictorial Data," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1991, pp. 165–174.

[83] Takahashi, S., K. Miyashita, S. Matsuoka, and A. Yonezawa, "A Framework for Constructing Animations via Declarative Mapping Rules," in *Proceedings of the IEEE Symposium on Visual Languages (VL)*, Oct. 1994, pp. 314–322.

[84] Tonouchi, T., K. Nakayama, S. Matsuoka, and S. Kawai, "Creating Visual Objects by Direct Manipulation," in *Proceedings of the IEEE Workshop on Visual Languages*, 1992, pp. 95–101.

[85] Tsang, E., *Foundations of Constraint Satisfaction.* London: Academic Press, 1993.

[86] Vander Zanden, B., "An Incremental Algorithm for Satisfying Hierarchies of Multi-Way Dataflow Constraints," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 1, Jan. 1996, pp. 30–72.

[87] Vander Zanden, B., G. A. Myers, D. Giuse, and P. Szekely, "The Importance of Pointer Variables in Constraint Models," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, Nov. 1991, pp. 155–164.

[88] Vander Zanden, B. T., "Incremental Constraint Satisfaction and Its Application to Graphical Interfaces," Tech. Rep. TR 88-941, Department of Computer, Science, Cornell University, Oct. 1988.

[89] Wilk, M. R., "Equate: An Object-Oriented Constraint Solver," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1991, pp. 286–298.

[90] Wilson, M., "Hierarchical Constraint Logic Programming (Ph.D. Dissertation)," Tech. Rep. 93-05-01, Dept. of Computer Science and Engineering, University of Washington, May 1993.

[91] Wilson, M. and A. Borning, "Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison," in *Proceedings of the North American Conference on Logic Programming*, 1989.

[92] Wilson, M. and A. Borning, "Hierarchical Constraint Logic Programming," Tech. Rep. 93-01-02a, Dept. of Computer Science and Engineering, University of Washington, Jan. 1993.

[93] Wilson, M. and A. Borning, "Hierarchical Constraint Logic Programming," *Journal of Logic Programming*, vol. 16, no. 3/4, July/Aug. 1993, pp. 277–319.

[94] Wolf, A., "Transforming Ordered Constraint Hierarchies into Ordinary Constraint Systems," in *Over-Constrained Systems* (M. Jampel, E. Freuder, , and M. Maher, eds.), vol. 1106 of *Lecture Notes in Computer Science*, Springer-Verlag, Aug. 1996, pp. 171–187.