A Scalable Linear Constraint Solver for User Interface Construction

Hiroshi Hosobe

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
hosobe@nii.ac.jp

Abstract. This paper proposes an algorithm for satisfying systems of linear equality and inequality constraints with hierarchical strengths or preferences. Basically, it is a numerical method that incrementally obtains the LU decompositions of linear constraint systems. To realize this, it introduces a novel technique for analyzing hierarchical systems of linear constraints. In addition, it improves performance by adopting techniques that utilize the sparsity and disjointness of constraint systems. Based on this algorithm, the HiRise constraint solver has been designed and implemented for the use of constructing interactive graphical user interfaces. This paper shows that HiRise is scalable up to thousands of simultaneous constraints in real-time execution.

1 Introduction

Constraints have been widely recognized to be powerful in the construction of *graphical user interfaces* (GUIs). The main usage of constraints in GUIs is to lay out graphical objects. Once a programmer defines the geometric relationship of objects with constraints, a constraint solver will automatically maintain the relationship afterward. Therefore, the programmer will be freed from the burden of writing the code to manage the layout. It is effective especially when the layout is too complex for the programmer to specify with a simple loop or recursion. Apart from geometric layouts, constraints can be used, for example, to adjust the sizes of graphical objects to internal data, and also to manage the relationships between internal data.

A major subject of the research on constraints for GUIs is how to model and solve various over-constrained real-world problems. For this purpose, *constraint hierarchies* [3] are often used as a theoretical framework. By definition, a constraint hierarchy is a constraint system that consists of constraints with hierarchical *strengths*, which can be regarded as the preferences or priorities of the constraints. Intuitively, (optimal) solutions of constraint hierarchies are determined so that they will satisfy as many strong constraints as possible, leaving weaker inconsistent constraints unsatisfied.

Another issue of the study on constraints for GUIs is how to improve the *scalability* of constraint satisfaction. For this purpose, incremental local propagation algorithms have been extensively explored [6, 9, 14, 15]. However, since

local propagation is basically limited to dataflow (or functional) equality constraints, the resulting algorithms inevitably impose restrictions on the kinds of possible constraint problems. A typical hurdle is to solve arbitrary hierarchies of linear equality and inequality constraints, although such constraint hierarchies arise naturally in actual GUI applications.

To address these issues, this paper proposes an algorithm for satisfying hierarchies of linear equality and inequality constraints. Its main contributions are summarized as follows:

- By introducing a novel technique called *hierarchical independence analysis*, it efficiently realizes the incremental satisfaction of hierarchies of linear equality constraints based on LU decomposition.
- It provides a way of handling linear inequality constraints in combination with quasi-linear optimization.
- It presents techniques for improving performance by utilizing the *sparsity* and *disjointness* of constraint hierarchies.

Based on this algorithm, the HiRise[1] constraint solver has been designed and implemented for the use of constructing interactive GUIs. Particularly, it is fitted to large-scale diagrams defined with numerous constraints. This paper shows that HiRise is scalable up to thousands of simultaneous constraints in real-time execution.

2 Related Work

There have been various algorithms proposed for solving constraint hierarchies. Particularly, in the area of GUIs, local propagation algorithms for dataflow equality constraints have been extensively studied. DeltaBlue [6] is the first efficient incremental algorithm in this category. SkyBlue [14], the successor of DeltaBlue, copes with the multiple outputs and cyclic dependencies of constraints. QuickPlan [15] ensures that it can solve a constraint hierarchy by local propagation if the hierarchy has at least one acyclic solution. The author proposed the *DETAIL* algorithm [9], which accommodated local propagation to the least-squares method as well as cyclic dependencies.

For the purpose of GUIs, there have been algorithms that deal with constraint hierarchies including inequalities in limited ways. Indigo [2] efficiently handles hierarchies with nonlinear inequality constraints by interval propagation, although it does not cope with the cyclic dependencies of constraints. The algorithm using projection [7] statically compiles constraint hierarchies with inequalities into program code.

The recent algorithms for solving constraint hierarchies with inequalities are Cassowary [1, 4] and QOCA [4, 10]. Both of the algorithms solve hierarchies of linear constraints by converting them into optimization problems. Cassowary uses the simplex method to obtain solutions based on the weighted sums of

[1] HiRise stands for 'HieRarchical linear system engine.'

constraint errors, while QOCA exploits linear complementary pivoting to find least-squares solutions.[2] Later, Section 8 provides further discussions about these algorithms.

Linear constraint satisfaction has also been studied in the community of constraint logic programming [13]. Its main issue is how to enhance the incremental satisfaction of ordinary (non-hierarchical) systems of linear equality and inequality constraints. By contrast, this paper primarily focuses on the incremental maintenance of hierarchical systems of linear constraints.

3 The Basic Algorithm

This section presents the basic algorithm for HiRise to satisfy hierarchical systems of linear equality constraints.

3.1 Problem Formulation

The basic algorithm focuses on linear equality constraints only. Instead of ordinary constraint hierarchies, it internally treats constraint systems formulated as follows:

Definition 1 (constraint system). A constraint system is an ordered set of m linear equations on n variables $x_1, x_2, \ldots, x_n$, where the i-th equation is represented as $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = c_i$. The *coefficient vector* of the i-th equation is $\boldsymbol{a}_i = (a_{i1}\ a_{i2}\ \cdots\ a_{in})$, and the *coefficient matrix* A, *variable vector* $\boldsymbol{x}$, and *constant vector* $\boldsymbol{c}$ of the system are defined as follows:

$$A = \begin{pmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{a}_m \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}, \quad \boldsymbol{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}.$$

Intuitively, the first equation is the strongest, and the latter an equation, the weaker it is. The notion of strength is similar to that of constraint hierarchies; that is, an equation has absolute priority over latter ones in determining solutions, which is strictly defined later.

For simplicity, the rest of this paper writes a constraint system as $A\boldsymbol{x} = \boldsymbol{c}$. Also, for brevity, it obeys the following notation rules: it assumes that the numbers of constraints and variables are m and n respectively; writing i and j, it intends indices that range over 1 to m and over 1 to n respectively. It sometimes attaches primes or subscripts to these symbols.

Solutions of constraint systems are defined using an ordering:

[2] Precisely, the QOCA constraint solving toolkit also provides the Cassowary solver to handle linear inequality constraints [10]. However, for convenience, this paper refers to the algorithm based on linear complementary pivoting as QOCA.

Definition 2 (solution). Given a constraint system $A\boldsymbol{x} = \boldsymbol{c}$, its solution set is

$$S(A\boldsymbol{x} = \boldsymbol{c}) \equiv \{\boldsymbol{v} \in \boldsymbol{R}^n \mid \forall \boldsymbol{v'} \in \boldsymbol{R}^n . |A\boldsymbol{v} - \boldsymbol{c}| \leq_{\mathrm{lex}} |A\boldsymbol{v'} - \boldsymbol{c}|\}$$

where $\leq_{\mathrm{lex}}$ is the lexicographic ordering, i.e., $|A\boldsymbol{v} - \boldsymbol{c}| \leq_{\mathrm{lex}} |A\boldsymbol{v'} - \boldsymbol{c}|$ is

$$\begin{aligned} & |A\boldsymbol{v} - \boldsymbol{c}| =_{\mathrm{lex}} |A\boldsymbol{v'} - \boldsymbol{c}| \equiv \forall i . |\boldsymbol{a}_i \boldsymbol{v} - c_i| = |\boldsymbol{a}_i \boldsymbol{v'} - c_i| \\ \text{or} \quad & |A\boldsymbol{v} - \boldsymbol{c}| <_{\mathrm{lex}} |A\boldsymbol{v'} - \boldsymbol{c}| \equiv \exists i . \forall i' < i . |\boldsymbol{a}_{i'} \boldsymbol{v} - c_{i'}| = |\boldsymbol{a}_{i'} \boldsymbol{v'} - c_{i'}| \\ & \qquad\qquad \wedge |\boldsymbol{a}_i \boldsymbol{v} - c_i| < |\boldsymbol{a}_i \boldsymbol{v'} - c_i| \ . \end{aligned}$$

A solution set $S(A\boldsymbol{x} = \boldsymbol{c})$ means that a solution of $A\boldsymbol{x} = \boldsymbol{c}$ is $x_1 = v_1$, $x_2 = v_2$, $\ldots$, $x_n = v_n$ for $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)^{\mathrm{T}} \in S(A\boldsymbol{x} = \boldsymbol{c})$. This paper simply refers to such a vector $\boldsymbol{v}$ as a solution.

Intuitively, $\leq_{\mathrm{lex}}$ 'hierarchically' compares two error vectors, and the solution set of the given constraint system is the set of all the variable value vectors that result in the minimum error vectors in the sense of $\leq_{\mathrm{lex}}$. Therefore, constraint systems may be regarded as holding preferential constraints in the total order.

Unlike constraint hierarchies, constraint systems by these definitions have no levels that contain constraints with equal preferences. However, the author has proved a theorem [8] that they have a close relationship with constraint hierarchies consisting of linear equations and solved with the locally-error-better (LEB, also known as locally-metric-better) comparator [3]. Informally, LEB determines the appropriateness of potential solutions based on how much each constraint is satisfied (see Section 8 for the discussion on comparators). The proved theorem means that the system obtained by 'serializing' (or putting in the total order) the constraints in each level of a hierarchy will always yield a subset of the LEB solution set of the original hierarchy. Thus, if all solutions are not necessary, Definitions 1 and 2 substitute for constraint hierarchies. Such a situation is common in various applications including GUIs that usually need only one solution. Therefore, the author believes that the notion of such constraint systems is useful as an alternative method to handle constraint hierarchies.

3.2 Hierarchical Independence

This subsection presents the notion of *hierarchical independence*, a foundation for analyzing hierarchical systems of linear constraints. It is the key technology for the basic algorithm to achieve efficiency.

A row of a coefficient matrix is said to be hierarchically independent if and only if it is linearly independent of all the upper (or stronger) ones:

Definition 3 (hierarchical independence). Given a constraint system $A\boldsymbol{x} = \boldsymbol{c}$, the condition that the i-th row of A is hierarchically independent, denoted as $\mathrm{hindep}(A, i)$, is defined as follows:

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

Also, a row that is not hierarchically independent is said to be hierarchically dependent.

The following theorem shows that solutions of constraint systems can be obtained by collecting and satisfying all the constraints corresponding to hierarchically independent rows:

Theorem 4. *For any constraint system $A\boldsymbol{x} = \boldsymbol{c}$, $\boldsymbol{v}$ is its solution if and only if*

$$\forall i.\, \mathrm{hindep}(A, i) \Rightarrow \boldsymbol{a}_i \boldsymbol{v} = c_i \ .$$

Proof. See [8] □

Conversely, if an equation has a dependent coefficient vector, it exhibits either inconsistency that must be discarded, or redundancy that may be ignored.

The rest of this paper says, for conciseness, that a constraint is *active* if and only if its coefficient vector is hierarchically independent, and refers to non-active constraints as *inactive* constraints. Also, for a constraint system $A\boldsymbol{x} = \boldsymbol{c}$, it considers an ordinary linear system $B\boldsymbol{x} = \boldsymbol{d}$ that contains all the active constraints of $A\boldsymbol{x} = \boldsymbol{c}$ in some arbitrary order. Obviously, the solution set of $A\boldsymbol{x} = \boldsymbol{c}$ is equal to that of $B\boldsymbol{x} = \boldsymbol{d}$. It calls such B and $\boldsymbol{d}$ an *active coefficient matrix* and an *active constant vector* respectively.

3.3 Solving a Constraint System from Scratch

This subsection presents how the basic algorithm solves a constraint system from scratch. For simplicity, the following description assumes that the given constraint system has n active constraints; for any system, it can be realized simply by adding to each variable a very weak default stay constraint that tries to preserve its current value.

Generally, given a constraint system $A\boldsymbol{x} = \boldsymbol{c}$, the basic algorithm obtains an LU decomposition in the following form:

$$BT_1T_2 \cdots T_t = L \tag{1}$$

where B is an active coefficient matrix, each T_k is a transformation matrix described later, and L is a lower triangular matrix:

$$B = \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \vdots \\ \boldsymbol{b}_n \end{pmatrix}, \qquad L = \begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ l_{n1} & l_{n2} & & 1 \end{pmatrix}.$$

Intuitively, (1) means that a sequence $T_1T_2 \cdots T_t$ transforms B into L.

Once the algorithm computes (1), it can solve $B\boldsymbol{x} = \boldsymbol{d}$ for $\boldsymbol{x}$ efficiently. First, it resolves $L\boldsymbol{y} = \boldsymbol{d}$ by computing $y_j \leftarrow d_j - \sum_{j'=1}^{j-1} l_{jj'} y_{j'}$, which is known as forward substitution [12]. Then it obtains $\boldsymbol{x}$ by calculating $\boldsymbol{x} = T_1T_2 \cdots T_t\boldsymbol{y}$.

When the basic algorithm solves $A\boldsymbol{x} = \boldsymbol{c}$ from scratch, it obtains an LU decomposition in the form:

$$BP_1U_1P_2U_2 \cdots P_nU_n = L$$

where each P_j and U_j is a permutation matrix and an upper triangular eta matrix respectively:[3]

$$P_j = \begin{pmatrix} E_{j-1} & & & \\ & 0 & & 1 \\ & & E_{j'-j-1} & \\ & 1 & & 0 \\ & & & & E_{n-j'} \end{pmatrix}, \quad U_j = \begin{pmatrix} E_{j-1} & & \\ & \frac{1}{b'_{jj}} & -\frac{b'_{j,j+1}}{b'_{jj}} \cdots -\frac{b'_{jn}}{b'_{jj}} \\ & & E_{n-j} \end{pmatrix}. \quad (2)$$

To determine which row to select as $\boldsymbol{b}_j$, the algorithm performs hierarchical independence analysis: until the j-th step, it has obtained a 'partial' LU decomposition as

$$\begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \vdots \\ \boldsymbol{b}_{j-1} \end{pmatrix} P_1U_1P_2U_2\cdots P_{j-1}U_{j-1} = \begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ l_{j-1,1} & \cdots & l_{j-1,j-2} & 1\ 0\cdots 0 \end{pmatrix}.$$

Then it picks up the uppermost unprocessed row $\boldsymbol{a}_i$ from A, and multiplies it by $P_1U_1P_2U_2\cdots P_{j-1}U_{j-1}$:

$$\boldsymbol{a}_iP_1U_1P_2U_2\cdots P_{j-1}U_{j-1} = (a'_{i1}\ a'_{i2}\ \cdots\ a'_{i,j-1}\ a'_{ij}\ \cdots\ a'_{in}).$$

If $a'_{ij'}$ is not a zero for some $j' \geq j$, $\boldsymbol{a}_i$ is a hierarchically independent row of A; then the algorithm assigns $\boldsymbol{a}_i$ to $\boldsymbol{b}_j$, and determines P_j and U_j with (2). Otherwise, it moves to the next uppermost unprocessed row since $\boldsymbol{a}_i$ is hierarchically dependent. Intuitively, P_j swaps the j-th column for the j'-th so that the resulting (j,j)-entry is not a zero. Then U_j changes the (j,j)-entry into one, and also 'eliminates' all the (j,j'')-entries for $j'' > j$.

The algorithm may be easily understood when compared with Gaussian elimination [12]: it transforms a matrix B into a lower triangular matrix L, while Gaussian elimination transforms a matrix into an upper triangular matrix. Unlike Gaussian elimination, it records its transformation process as a sequence of P_j's and U_j's.

In addition to ordinary linear equality constraints, the algorithm can handle *edit* and *stay* constraints, which usually arise in GUI applications: an edit constraint for a variable attempts to change its value, while a stay constraint tries to fix the value of the designated variable. The algorithm realizes an edit constraint by expressing it as $x = c$ and calculating the necessary parts of $L\boldsymbol{y} = \boldsymbol{d}$ and $\boldsymbol{x} = T_1T_2\cdots T_t\boldsymbol{y}$ whenever it tries to alter the value of x. By contrast, the algorithm implements a stay constraint simply by representing it as $x = c$ where c indicates the value of x immediately before it becomes active.

The time complexity for constructing an LU decomposition from scratch is $O(mn^2)$. Also, the time complexity for calculating variable values using the LU decomposition is $O(n^2)$.

[3] In this paper, E_k represents the $k \times k$ identity matrix.

3.4 Inserting a Constraint Incrementally

When a new constraint is inserted between constraints in the current system, the basic algorithm takes one of the following actions:

- If it needs to activate the new constraint, it updates the current LU decomposition. To do this, it first finds an appropriate 'victim' constraint that should be deactivated instead. After eliminating the row for the victim and appending the row for the new one, it revises the LU decomposition.
- Otherwise, it keeps the present LU decomposition unchanged.

The criterion for activating or deactivating a constraint is its hierarchical independence in the new system. The victim is the constraint that has been active in the previous system but becomes inactive in the new system because of the stronger, inserted constraint. When a constraint is inserted, there will be at most one victim because all the constraints are linear equations.

The technique for updating LU decompositions is inspired by Forrest-Tomlin method [5], which was originally devised for linear programming. The technique is as follows: assume that the new constraint $\boldsymbol{a}\boldsymbol{x} = c$ should be activated, and that the row for the victim is found to be the j-th one of the active coefficient matrix B, which has been decomposed into (1). Let

$$B' = \begin{pmatrix} \boldsymbol{b}_1 \\ \vdots \\ \boldsymbol{b}_{j-1} \\ \boldsymbol{b}_{j+1} \\ \vdots \\ \boldsymbol{b}_n \end{pmatrix}, \qquad Q_j = \left(\begin{array}{c|c|c} E_{j-1} & & \\ \hline & & 1 \\ \hline & E_{n-j} & \end{array}\right)$$

and then the following holds:

$$B'T_1T_2\cdots T_tQ_j = \begin{pmatrix} 1 & & & & & & \\ \vdots & \ddots & & & & & \\ l_{j-1,1} & & 1 & & & & \\ l_{j+1,1} & \cdots & l_{j+1,j-1} & 1 & & l_{j+1,j} & \\ \vdots & & \vdots & & \ddots & \vdots & \\ l_{n1} & \cdots & l_{n,j-1} & l_{n,j+1} & 1 & l_{nj} & \end{pmatrix}.$$

Thus, all the entries at the rightmost column can be eliminated with appropriate matrices U'_j, U'_{j+1}, …, U'_{n-1} in the form (2). Then, with U'_n such that

$$\boldsymbol{a}T_1T_2\cdots T_tQ_jU'_jU'_{j+1}\cdots U'_{n-1}U'_n = (l_1\ l_2\ \cdots\ l_{n-1}\ 1)$$

the following new LU decomposition is obtained:

$$\begin{pmatrix} \boldsymbol{b}_1 \\ \vdots \\ \boldsymbol{b}_{j-1} \\ \boldsymbol{b}_{j+1} \\ \vdots \\ \boldsymbol{b}_n \\ \boldsymbol{a} \end{pmatrix} T_1\cdots T_tQ_jU'_j\cdots U'_{n-1}U'_n = \begin{pmatrix} 1 & & & & & \\ \vdots & \ddots & & & & \\ l_{j-1,1} & & 1 & & & \\ l_{j+1,1} & \cdots & l_{j+1,j-1} & 1 & & \\ \vdots & & \vdots & & \ddots & \\ l_{n1} & \cdots & l_{n,j-1} & l_{n,j+1} & 1 & \\ l_1 & \cdots & l_{j-1} & l_j & l_{n-1} & 1 \end{pmatrix}.$$

To judge whether to activate the new constraint and (if necessary) which active constraint to victimize, the algorithm carries out hierarchical independence analysis: it finds the row index j for the first active constraint, in the descending order of the index (or preference) i of the constraint, such that

$$\boldsymbol{a}T_1T_2\cdots T_tQ_jU'_jU'_{j+1}\cdots U'_{n-1} = (a'_1\ a'_2\ \cdots\ a'_{n-1}\ a'_n)$$

where $a'_n \neq 0$ and U'_j, U'_{j+1}, ..., U'_{n-1} are the ones obtained with the above technique. If such i is no smaller than the index where the new constraint is inserted, the algorithm needs to revise the LU decomposition, and the i-th constraint is the victim. Otherwise, it should not change the current decomposition.

The time complexity for inserting a constraint into a system is $O(m'n^2)$, where m' indicates the number of the constraints tested for a victim. Usually $m' = 1$ holds since the weakest active constraint tends to be the victim.

3.5 Deleting a Constraint Incrementally

When an existing constraint is deleted from the current constraint system, the basic algorithm applies one of the following processes:

- If the deleted constraint is active in the present LU decomposition, it updates the decomposition. To do this, it first eliminates the row for the deleted one by the method described in the previous subsection. Then it detects a proper alternative constraint that should be activated instead, and updates the decomposition by attaching the row for the alternative.
- Otherwise, it preserves the current LU decomposition.

To decide which inactive constraint $\boldsymbol{a}_i\boldsymbol{x} = c_i$ to activate alternatively, the algorithm employs hierarchical independence analysis: assume that j is the row index in B of the deleted constraint, and that U'_j, U'_{j+1}, ..., U'_{n-1} are obtained with the elimination of the j-th row; then it searches for the index (or preference) i of the first inactive constraint, in the ascending order of i, such that

$$\boldsymbol{a}_iT_1T_2\cdots T_tQ_jU'_jU'_{j+1}\cdots U'_{n-1} = (a'_{i1}\ a'_{i2}\ \cdots\ a'_{i,n-1}\ a'_{in})$$

where $a'_{in} \neq 0$. With such an alternative constraint, the algorithm obtains the new LU decomposition.

The time complexity for deleting a constraint from a system is $O(m'n^2)$, where m' represents the number of the constraints tried for an alternative. If the system does not have many conflicting (or possibly redundant) constraints, m' will be bounded by a small number.

4 Handling Inequalities

This section provides a functionally enhanced algorithm for handling inequality constraints. First, it solves a given constraint system in the same way as the basic algorithm. Then it collects all the inactive constraints and resolves them with

quasi-linear optimization, which is similar to the Cassowary constraint solver [1, 4]. Quasi-linear optimization finds a vector that satisfies a set of linear equality and inequality constraints and also that minimizes an objective function composed as a sum of absolute values of linear expressions.

The enhanced algorithm treats both equality constraints $\boldsymbol{a}\boldsymbol{x} = c$ and inequality ones $\boldsymbol{a}\boldsymbol{x} \geq c$. However, in the first step, it assumes that all the constraints were equations, and obtains an LU decomposition (1) in the same way as the basic algorithm.

In the second step, it correctly adjusts the constraints by introducing what are called 'slack variables' in the area of linear programming. It rewrites each constraint into $\boldsymbol{a}\boldsymbol{x} = c + \sigma s$ where $s \geq 0$, and $\sigma = 0$ if the constraint is an equation, or $\sigma = 1$ if not. With the rewriting, the constraint system can be expressed as $A\boldsymbol{x} = \boldsymbol{c} + F\boldsymbol{s}$ where $\boldsymbol{s} = (s_1, s_2, \ldots, s_m)^{\mathrm{T}} \geq \boldsymbol{0}$ and F is the $m \times m$ diagonal matrix whose (i, i)-th entry is σ_i $(= 0$ or $1)$.

Next, the algorithm represents each inactive constraint $\boldsymbol{a}_i\boldsymbol{x} = c + \sigma_i s_i$ with only $\boldsymbol{s}$. To do this, it expresses $\boldsymbol{x}$ with $\boldsymbol{s}$ (more precisely, s_i's for the active constraints) by introducing an $n \times m$ matrix G that consists of the rows of F corresponding to the active constraints. Then, solving $L\boldsymbol{y} = \boldsymbol{d} + G\boldsymbol{s}$ and $\boldsymbol{x} = T_1 T_2 \cdots T_t \boldsymbol{y}$, it obtains $\boldsymbol{x} = \boldsymbol{v} + H\boldsymbol{s}$. Thus it rewrites the inactive constraint into $\boldsymbol{a}_i H\boldsymbol{s} - \sigma_i s_i = c_i - \boldsymbol{a}_i \boldsymbol{v}$.

Finally, to obtain the values of $\boldsymbol{s}$, the algorithm creates a quasi-linear optimization problem and resolves it with the simplex method [12]. For the indices i_1, i_2, …, $i_{m'}$ of the inactive constraints, it constructs the following problem by introducing new non-negative variables $\delta^+_{i_k}$ and $\delta^-_{i_k}$ for $1 \leq k \leq m'$:

$$\begin{aligned} &\text{minimize} && \textstyle\sum_{k=1}^{m'} \left\{ w_{i_k} (\delta^+_{i_k} + \delta^-_{i_k}) \right\} \\ &\text{subject to} && \boldsymbol{a}_{i_k} H\boldsymbol{s} - \sigma_{i_k} s_{i_k} = c_{i_k} - \boldsymbol{a}_{i_k}\boldsymbol{v} + \delta^+_{i_k} - \delta^-_{i_k} \quad (1 \leq k \leq m') \end{aligned} \tag{3}$$

where w_{i_k} indicates the weight corresponding to the preference of the i_k-th constraint (e.g. 10^6, 10^3, and 1 for strong, medium, and weak constraints respectively).[4] Intuitively, each $(\delta^+_{i_k} + \delta^-_{i_k})$ indicates the error $|\boldsymbol{a}_{i_k} H\boldsymbol{s} - \sigma_{i_k} s_{i_k} - c_{i_k} + \boldsymbol{a}_{i_k}\boldsymbol{v}|$ of the k-th inactive constraint, and the problem minimizes the sum of the weighted errors of the inactive constraints.

To understand the enhanced algorithm, consider the constraint system consisting of $x_1 \geq 0$, $x_2 \geq 0$, $-x_1 \geq -2$, $-x_2 \geq -2$, and $x_1 + x_2 = 5$ in this order, and also assume that their weights are 10^6, 10^6, 10^3, 10^3, and 1 respectively. First, the algorithm selects $x_1 \geq 0$ and $x_2 \geq 0$ as the active constraints, and constructs an LU decomposition for $x_1 = 0$ and $x_2 = 0$. Second, it introduces non-negative variables s_1, s_2, s_3, s_4, and s_5, and rewrites all the constraints into $x_1 = 0 + s_1$, $x_2 = 0 + s_2$, $-x_1 = -2 + s_3$, $-x_2 = -2 + s_4$, and $x_1 + x_2 = 5 + 0 \cdot s_5$ respectively. Next, using the LU decomposition, it expresses x_1 and x_2 as $x_1 = s_1$ and

[4] It should be noted that such real-valued weights might lead to incorrect solutions. In fact, Cassowary avoids this problem by introducing 'symbolic' weights [1]. However, HiRise is optimistic about the problem since it is assumed to handle a relatively small number of inequality conflicting constraints (see Section 8).

$x_2 = s_2$. Then it rewrites the remaining inactive constraints into $-s_1 - s_3 = -2$, $-s_2 - s_4 = -2$, and $s_1 + s_2 - 0 \cdot s_5 = 5$. Finally, it resolves the optimization problem that minimizes $10^3(\delta_3^+ + \delta_3^-) + 10^3(\delta_4^+ + \delta_4^-) + (\delta_5^+ + \delta_5^-)$ subject to $-s_1 - s_3 = -2 + \delta_3^+ - \delta_3^-$, $-s_2 - s_4 = -2 + \delta_4^+ - \delta_4^-$, and $s_1 + s_2 - 0 \cdot s_5 = 5 + \delta_5^+ - \delta_5^-$. Any solution to this problem must satisfy $s_1 = s_2 = 2$, $s_3 = s_4 = \delta_3^+ = \delta_3^- = \delta_4^+ = \delta_4^- = \delta_5^+ = 0$, and $\delta_5^- = 1$ (s_5 may be arbitrary). Thus the solution to the original system is $x_1 = 2$ and $x_2 = 2$. Note that the weakest constraint $x_1 + x_2 = 5$ is maximally satisfied.

Similar to the basic algorithm, the enhanced algorithm as a whole can be regarded as handling the locally-error-better (LEB) comparator. It is because the objective function (3) implements weighted-sum-better (WSB), which is more strictly restrictive than LEB; that is, any WSB solution is also an LEB one [3].

If edit constraints try to change variable values, the algorithm updates $c_{i_k} - \boldsymbol{a}_{i_k}\boldsymbol{v}$ for each k (which also reflects stay constraints), and then incrementally re-optimizes the problem in the same way as Cassowary. If inserted or deleted constraints update the LU decomposition, it reconstructs a problem.

Usually, the algorithm can considerably reduce the sizes of optimization problems as follows:

- If the i-th constraint is equality ($\sigma_i = 0$), it may eliminate s_i. In the above example, s_5 could be deleted.
- If the k-th inactive constraint is inequality ($\sigma_{i_k} = 1$), it can omit $\delta_{i_k}^+$. The above example could remove δ_3^+ and δ_4^+.

With these reductions, the numbers of variables and constraints in an optimization problem become $(m_1 + m_2 + m - n)$ and $(m - n)$ respectively, where m_1 and m_2 indicate the numbers of the inequality constraints and inactive equality ones respectively.[5] Therefore, if the given system contains only small numbers of inequality constraints and conflicting (or possibly redundant) constraints, the algorithm can efficiently solve the optimization problem.

5 Performance Techniques

This section provides two techniques for improving the performance of the basic algorithm. Both of them can be used together with the technique for handling inequality constraints.

5.1 Utilizing Sparsity

In usual GUI applications, constraint systems are *sparse*, that is, most individual constraints refer to only small numbers of variables even if entire systems are large. Therefore, resulting coefficient matrices are also sparse ones where most

[5] Note that the simplex method needs not to introduce artificial variables. For each rewritten inactive constraint, it can always select one of $\delta_{i_k}^+$, $\delta_{i_k}^-$, and s_{i_k} as a basic variable.

entries are zeros. From the viewpoint of efficiency, it is desirable for the algorithm to preserve the sparsity of the matrices. However, in transforming them into lower triangular ones, it sometimes yields nonzero entries called 'fill-ins' [11] at the positions where nonzero entries have been lain. Thus it may degrade the sparsity of the original matrices.

This subsection describes a performance technique using the sparsity of constraint systems. It restrains the occurrences of fill-ins by adopting an ordering method for sparse matrices [11]. Generally, ordering methods exchange rows and columns of matrices to minimize the numbers of fill-ins.

To realize this, the proposed technique separates constraints in a system into required and preferential ones, and applies an ordering method to the partial matrix corresponding to required constraints. This process is possible because there are no differences among the preferences of required constraints.

The technique performs kernel generation and Tewarson's method [11] below while the basic algorithm is computing an LU decomposition from scratch.

1. Generate a kernel in the following two steps:
   (a) Seek a required constraint with only one variable. If such a constraint is found, move its nonzero entry to the pivot position, transform it into one, and repeat this operation (ignore the processed variable afterward).
   (b) Search for a required constraint with a variable referred by no other required ones. If such a constraint is detected, move the corresponding entry to the pivot, alter it into one, eliminate the remaining entries, and iterate this operation (disregard this constraint afterward).
2. Perform LU decomposition successively for the rest of the required constraints. Use Tewarson's method in selecting variables and constraints for pivoting; that is, by examining the partial lower triangular matrix obtained by the current LU decomposition, minimize the product $\{$(the number of the unprocessed nonzero entries in the row corresponding to the constraint) $- 1\} \times \{$(the number of the unprocessed nonzero entries in the column corresponding to the variable) $- 1\}$.

For some situations, handling the sparsity of preferential constraints might be promising. The above technique is also applicable to the inside of each preferential level of constraint hierarchies, although it has not been implemented.

5.2 Exploiting Disjointness

In many GUI applications, constraint systems have *disjointness*; that is, large systems may be divided into multiple smaller independent components. For efficiency, it is preferable for the algorithm to solve such disjoint components separately.

This subsection explains a performance technique adopting the disjointness of constraint systems. Basically, for each disjoint component of a system, it maintains and solves a distinct subsystem.

It is necessary to integrate LU decompositions when added constraints merge multiple components. For simplicity, suppose that two components need to be

merged. Since they are disjoint, they do not share any variables. Therefore, even if they are merged, the hierarchical independence of each constraint will not change. Thus, with the active constraint sets $B\boldsymbol{x} = \boldsymbol{c}$ and $B'\boldsymbol{x'} = \boldsymbol{c'}$ for these components, the merged set can be expressed as follows:

$$\begin{pmatrix} B & 0 \\ 0 & B' \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x'} \end{pmatrix} = \begin{pmatrix} \boldsymbol{c} \\ \boldsymbol{c'} \end{pmatrix}. \tag{4}$$

Now, let the LU decompositions of B and B' be $BT_1T_2 \cdots T_t = L$ and $B'T'_1T'_2 \cdots T'_{t'} = L'$ respectively. Then the following holds:

$$\begin{pmatrix} B & 0 \\ 0 & B' \end{pmatrix} \begin{pmatrix} T_1 & 0 \\ 0 & E \end{pmatrix} \begin{pmatrix} T_2 & 0 \\ 0 & E \end{pmatrix} \cdots \begin{pmatrix} T_t & 0 \\ 0 & E \end{pmatrix} \begin{pmatrix} E & 0 \\ 0 & T'_1 \end{pmatrix} \begin{pmatrix} E & 0 \\ 0 & T'_2 \end{pmatrix} \cdots \begin{pmatrix} E & 0 \\ 0 & T'_{t'} \end{pmatrix} = \begin{pmatrix} L & 0 \\ 0 & L' \end{pmatrix}.$$

It can be regarded as an LU decomposition of the active coefficient matrix of (4) in the form of (1). Thus multiple separately solved components can be efficiently integrated.

It might be more fruitful to utilize the 'partial' disjointness of constraint systems that consist of almost disjoint components. The current (perhaps unsatisfactory) solution to this issue is to recursively apply the above disjointness technique, which actually depends on the way of the construction of constraint systems. Another more aggressive solution is an open problem.

6 The HiRise Constraint Solver

Based on the proposed algorithm, Java and C++ versions of the HiRise constraint solver have been developed. Mainly, HiRise was designed for the construction of interactive GUIs. It allows programmers to create variables and constraints as Java or C++ objects, and to insert/delete constraints into/from the solver object. The supported kinds of constraints are linear equality, linear inequality, stay, and edit. Currently, the Java version provides the full functionality of HiRise, whereas the C++ version implements only the basic algorithm. The present Java implementation consists of approximately ten thousand lines of code. Fig. 1 illustrates the screen snapshots of sample applications developed in C++ for Microsoft Windows.

7 Experiments

This section provides the results of two experiments on the performance of the HiRise constraint solver. Both of the experiments used an actual application for editing a tree depicted in Fig. 2. In the application, the layout of a tree is defined with constraints as follows: subtrees sharing the same parent nodes are adjacent, and the intervals of neighboring leaves are equal. Also, it adds six inequality constraints to a tree: four inequalities confine the tree in the window, and the other two prevent it from getting reversed. The application automatically generates a tree with an irregular structure using random numbers.

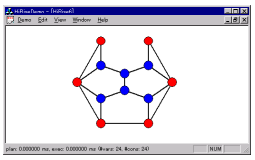
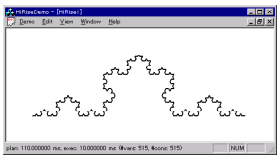
(a)  (b) 

Fig. 1. Sample applications of HiRise: (a) one that allows a user to edit a graph by adding, moving, and staying nodes, and fixing edge directions, where each inner node is constrained at the barycenter of its adjacent nodes; (b) another that enables a user to operate a picture that approximates the fractal diagram known as the Koch curve, which is realized with a finite number of vertices constrained by linear equations

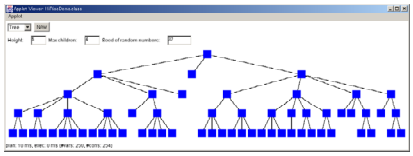


Fig. 2. An application for editing a tree

First, Experiment 1 compared the performance of HiRise with Cassowary [1, 4] and QOCA [4, 10] using medium-scale constraint systems. The used implementations of Cassowary and QOCA were version 0.55 in Java and version 1.0 beta 2 of LinIneqSolver in Java respectively, both of which were distributed by the authors of the original papers. These programs were compiled and executed with Java Development Kit 1.2.1 from Sun Microsystems. The execution environment was a Sun Ultra 60 workstation with a single 296 MHz UltraSPARC-II processor running Solaris 7.

The table below shows the results of Experiment 1. It gives the numbers of inserted and deleted constraints, the total numbers of constraints, and the times in milliseconds required for executing operations of editing a tree. In the experiment, each solver was given a tree of the same shape generated with a certain seed of random numbers.

	Numbers of constraints			Times for execution		
	Insert	Delete	Total	HiRise	Cassowary	QOCA
Initial layout	512	4	508	1771	2289	4953
Start move	2	0	510	8	1316	34
Repeat move	0	0	510	1	2	3
Finish move	0	2	508	8	1485	1
Add node	6	2	512	28	117	741
Remove node	2	6	508	26	103	122

Overall, HiRise exhibited higher performance than Cassowary and QOCA. In particular, its incremental constraint satisfaction is usually much faster than them, which impresses the power of hierarchical independence analysis.

Next, Experiment 2 measured the scalability of HiRise. The following table illustrates the results of this experiment, where the numbers of constraints are the ones immediately after the initial solutions were computed:

Numbers of constraints	508	1024	1524	2012	2536
Initial layout	1771	12777	38634	87161	170062
Start move	8	15	23	31	40
Repeat move	1	3	4	5	6
Finish move	8	13	21	28	35
Add node	28	148	265	452	694
Remove node	26	114	189	245	380

The results indicate that HiRise is sufficiently rapid even for a system of more than two thousand constraints. The only problem is that it costs much times to obtain initial solutions, which is $O(mn^2)$ in time complexity. However, the author is attempting to alleviate this problem since the current implementation for utilizing the sparsity of constraint systems is rather naive.

It should be noted that the performance of HiRise actually depends on various aspects of its algorithm, that is, hierarchical independence analysis, the way of inequality handling, and the performance techniques for sparsity and disjointness. A more thorough evaluation of how much each of them works for different situations is one of the future work.

8 Discussion

As shown in the experimental results, HiRise is usually faster than Cassowary [1, 4] and QOCA [4, 10] for hundreds of constraints, and is further scalable up to thousands of constraints. It is because HiRise performs hierarchical independence analysis for fast handling preferential constraints, and also because it adopts the performance techniques using the sparsity and disjointness of constraint systems. Only, it may slow down as the number of inequalities grows, since it must reconstructs an internal simplex tableau after updating the corresponding LU decomposition.

To solve constraint hierarchies, HiRise uses the locally-error-better (LEB) comparator, which is the same as Indigo [2]. By contrast, Cassowary adopts weighted-sum-better, which is a little more restrictive than LEB, and QOCA exploits least-squares-better (LSB), which is further more discriminative than LEB. It is known that LSB is useful to applications with many conflicting preferential constraints, because it relaxes the constraints by uniformly distributing their errors and thus exhibits the 'least-surprise' behavior to users. Therefore, QOCA is sometimes the most functionally advantageous among these solvers.

In summary, HiRise is suitable for massive constraint systems including relatively small numbers of inequalities and not necessitating the uniform relaxation of conflicts. Particularly, it is fitted to properly designed large-scale diagrams, as proved in the previous section.

9 Conclusions and Future Work

This paper proposed an algorithm for satisfying systems of linear equality and inequality constraints with hierarchical preferences. It also presented the HiRise constraint solver, which is based on the algorithm and is designed for user interface construction, and it showed that HiRise is scalable up to thousands of simultaneous constraints in real-time execution.

Using HiRise, the author is developing a Java-based constraint programming language that allows programmers to specify constraints more easily. Also, the author is planning to revise the C++ version of HiRise so that it will provide the full functionality and also a further scalability.

Acknowledgments

The author would like to thank the anonymous referees for their insightful reviews. This research was supported in part by the Japan Society for the Promotion of Science, Grant-in-Aid for Encouragement of Young Scientists, 12780252, 2000.

References

[1] Badros, G. J., Borning, A.: The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Tech. Rep. 98-06-04, Dept. of Computer Science and Engineering, Univ. of Washington, 1998.
[2] Borning, A., Anderson, R., Freeman-Benson, B.: Indigo: A local propagation algorithm for inequality constraints. In *Proc. ACM UIST*, pp. 129–136, 1996.
[3] Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.
[4] Borning, A., Marriott, K., Stuckey, P., Xiao, Y.: Solving linear arithmetic constraints for user interface applications. In *Proc. ACM UIST*, pp. 87–96, 1997.
[5] Chvátal, V.: *Linear Programming*. Freeman, 1983.
[6] Freeman-Benson, B. N., Maloney, J., Borning, A.: An incremental constraint solver. *Comm. ACM*, 33(1):54–63, 1990.
[7] Harvey, W., Stuckey, P., Borning, A.: Compiling constraint solving using projection. In *Principles and Practice of Constraint Programming—CP97*, vol. 1330 of *LNCS*, pp. 491–505. Springer, 1997.
[8] Hosobe, H.: *Theoretical Properties and Efficient Satisfaction of Hierarchical Constraint Systems*. PhD thesis, Dept. of Information Science, Univ. of Tokyo, 1998.
[9] Hosobe, H., Miyashita, K., Takahashi, S., Matsuoka, S., Yonezawa, A.: Locally simultaneous constraint satisfaction. In *Principles and Practice of Constraint Programming—PPCP'94*, vol. 874 of *LNCS*, pp. 51–62. Springer, 1994.
[10] Marriott, K., Chok, S. C., Finlay, A.: A tableau based constraint solving toolkit for interactive graphical applications. In *Principles and Practice of Constraint Programming—CP98*, vol. 1520 of *LNCS*, pp. 340–354. Springer, 1998.
[11] Oguni, C., Murata, K., Miyoshi, T., Dongarra, J. J., Hasegawa, H.: *Matrix Computing Software*. Maruzen, 1991. In Japanese.
[12] Press, W. H., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T.: *NUMERICAL RECIPES in C: The Art of Scientific Computing*. Cambridge University Press, 1988.
[13] Refalo, P., Hentenryck, P. V.: CLP(R_{lin}) revised. In *Proc. JICSLP*, pp. 22–36. MIT Press, 1996.
[14] Sannella, M.: SkyBlue: A multi-way local propagation constraint solver for user interface construction. In *Proc. ACM UIST*, pp. 137–146, 1994.
[15] Vander Zanden, B.: An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Prog. Lang. Syst.*, 18(1):30–72, 1996.