A Simplex-Based Scalable Linear Constraint Solver for User Interface Applications

Hiroshi Hosobe
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
e-mail: hosobe@acm.org

***Abstract*—We propose a scalable algorithm called HiRise2 for incrementally solving soft linear constraints over real domains. It is based on a framework for soft constraints, known as constraint hierarchies, to allow effective modeling of user interface applications by using hierarchical preferences for constraints. HiRise2 introduces LU decompositions to improve the scalability of an incremental simplex method. Using this algorithm, we implemented a constraint solver. We also show the results of experiments on the performance of the solver.**

***Keywords*-soft constraints; linear constraints; constraint solving; simplex method; user interfaces**

I. Introduction

Constraints provide a powerful tool for solving various problems, and are used in fields including programming and artificial intelligence. Especially in programming user interface applications such as a drawing editor, it is natural to express the layout of graphical objects by constraints, and the automatic solving of constraints eases the process of graphical layout. Therefore programming with constraints has been studied in the field of user interfaces since its infancy [1].

In programming constraint-based user interfaces, it is important to process *preferences* for constraints. Programmers need to impose various constraints such as ranges of object positions on the screen and dragging of an object with a mouse as well as positional relations among graphical objects; therefore it is often difficult for them to specify such constraints without inconsistencies. *Constraint hierarchies* [2] are widely used as a theoretical framework for processing preferences for constraints. In a constraint hierarchy, each constraint is associated with a hierarchical preference called a *strength*, and solutions are determined to satisfy as many strong constraints as possible.

In this paper we propose an algorithm called HiRise2 for solving hierarchies of *linear constraints* including linear inequalities. It can be regarded as an extension of Cassowary [3], [4], a variant of the *simplex method* for hierarchies of linear constraints. HiRise2 is different from Cassowary mainly in the following two points:

- It internally treats the problem as an *ordered constraint hierarchy* [5]–[7];
- It adopts *LU decomposition* for its simplex method.

HiRise2 is also similar to our previous algorithm called HiRise [8]–[11] in that both employ ordered constraint hierarchies and LU decomposition; however, HiRise uses them only for processing equality constraints. HiRise2 achieves efficient constraint solving by introducing these techniques into the whole process including inequality manipulation. Therefore HiRise2 can be viewed as an integration of the advantages of HiRise and Cassowary. In addition, HiRise2 enables efficient recomputation of solutions by *incremental* constraint satisfaction [12] as with HiRise and Cassowary.

The rest of this paper is organized as follows. After presenting related work in Section II, we provide preliminary introductions to constraint hierarchies in Section III. Then we propose the HiRise2 algorithm in Section IV, and describe its implementation in Section V and the results of experiments on its performance in Section VI. We describe conclusions and future work in Section VII.

II. Related Work

Most of the early algorithms for solving constraint hierarchies, such as DeltaBlue [12], [13], treat local propagation constraints by using a graph-theoretic approach. Such algorithms are not appropriate for simultaneous constraints that depend on each other, and therefore they are applicable only to a limited class of problems.

Linear constraint solving algorithms were developed to treat simultaneous constraints in constraint hierarchies. Cassowary [3], [4] solves hierarchies of linear equality and inequality constraints by transforming such a hierarchy into a linear programming problem and then by applying an incremental simplex method to the resulting problem. An extended version of Cassowary [14] processes disjunctions of linear constraints that are useful for representing non-overlap constraints. QOCA [3], [15], [16] solves hierarchies of linear constraints by using tableau-based algorithms. An old version of QOCA [3] uses the active set method to handle the least-squares criterion for constraint hierarchies. For this purpose, QOCA's QCLinIneqSolver [15], [16] performs linear complementary pivoting. HiRise [8]–[11] also solves hierarchies of linear constraints. Its initial versions [8], [9] treat only linear equality constraints by using an LU decomposition-based algorithm, and its later versions [10], [11] additionally support linear inequality constraints by externally combining a simplex method with its LU decomposition-based algorithm. The HiRise2 algorithm, which we propose in this paper, is along this line of research.

Researchers have been studying more complex constraints than local propagation and linear ones. In the field of user interfaces, for example, the Chorus constraint solver [17], which approximately processes hierarchies of nonlinear constraints, and a method that dynamically performs linear approximation of geometric constraints [18] were developed. In the field of computer-aided design, there has been much research on geometric constraint solving methods [19] that are to some extent similar to the aforementioned graph-theoretic approach to local propagation constraints. By contrast, the HiRise2 algorithm aims at efficient processing of hierarchies of linear constraints.

III. Preliminaries

This section provides preliminary introductions to constraint hierarchies and ordered constraint hierarchies.

A. Constraint Hierarchies

We first describe the formulation of constraint hierarchies [2]. Let $[i, j]$ denote the interval of integers from i to j. Let $\mathbf{x}$ be the vector $(x_1, x_2, \ldots, x_n)$ of n variables over the real number domain $\mathbb{R}$. Let $\mathbf{v}$ (possibly with primes) be the vector $(v_1, v_2, \ldots, v_n)$ of n real numbers (which indicates a variable assignment to $\mathbf{x}$). Let C be the set of all constraints (or each element in C can be regarded as a constraint identifier). A strength is an integer between 0 and h, where h is a positive integer. Intuitively, strength 0 is the strongest, and a larger number indicates a weaker strength. A constraint hierarchy H is an indexed finite multiset $\{c_{k,i}\}_{k\in[0,h]\wedge i\in[1,m_k]}$ of constraints. Intuitively, $c_{k,i}$ indicates the i-th constraint among the constraints associated with strength k. The k-th level of H, denoted as H_k, is the indexed multiset $\{c_{k,i}\}_{i\in[1,m_k]}$. The constraints in H_0 are said to be required, and the others are said to be preferential.

An error function is of type $C \times \mathbb{R}^n \to \mathbb{R}_{0+}$, where $\mathbb{R}_{0+}$ is the set of all nonnegative real numbers. Intuitively, $\mathrm{error}(c, \mathbf{v})$ returns the violation of c for the value $\mathbf{v} \in \mathbb{R}^n$ of $\mathbf{x}$; especially, it returns 0 if $\mathbf{v}$ satisfies c. Given a constraint hierarchy H, a comparator "better" is defined as a relation of type $\mathbb{R}^n \times \mathbb{R}^n$ to determine solutions to H. Intuitively, $\mathrm{better}(\mathbf{v}, \mathbf{v}')$ indicates whether $\mathbf{v}$ better satisfies preferential constraints in H than $\mathbf{v}'$.

A solution to H is such a variable value vector that there is no better vector among those satisfying all the required constraints. Formally, the set $S(H)$ of the solutions to H is defined as follows:

$$S(H) = \{\mathbf{v} \in S_0(H) \mid \neg\,\exists \mathbf{v}' \in S_0(H), \mathrm{better}(\mathbf{v}', \mathbf{v})\}$$

where

$$S_0(H) = \{\mathbf{v} \in \mathbb{R}^n \mid \forall c_{0,i} \in H_0, \mathrm{error}(c_{0,i}, \mathbf{v}) = 0\}.$$

Different comparators result in different kinds of solutions. An important comparator is locally-better, which is defined as

$$\begin{aligned}\text{locally-better}(\mathbf{v}, \mathbf{v}') \equiv\ & \exists k \in [1, h], \forall k' \in [1, k-1],\\ & (\forall c_{k',i} \in H_{k'}, \mathrm{error}(c_{k',i}, \mathbf{v}) = \mathrm{error}(c_{k',i}, \mathbf{v}')) \wedge\\ & (\forall c_{k,i} \in H_k, \mathrm{error}(c_{k,i}, \mathbf{v}) \le \mathrm{error}(c_{k,i}, \mathbf{v}')) \wedge\\ & (\exists c_{k,i} \in H_k, \mathrm{error}(c_{k,i}, \mathbf{v}) < \mathrm{error}(c_{k,i}, \mathbf{v}')).\end{aligned}$$

There are two locally-better comparators [2], locally-error-better (LEB; also known as locally-metric-better, LMB) [20] and locally-predicate-better (LPB) [12], and they use different error functions. LEB is known to be suited to constraint-based user interfaces.

B. Ordered Constraint Hierarchies

In general, locally-better allows many solutions. However, it is often sufficient for applications to find one locally-better solution. Also, it is known that user interface applications sometimes suffer from the "split stay" problem [7] due to the existence of multiple solutions.

Ordered constraint hierarchies [5], [6] are useful for such situations. Intuitively, in an ordered constraint hierarchy, preferences for constraints are totally ordered inside each level as well as among levels. Solutions to an ordered constraint hierarchy H are defined by the following ordered-better comparator:

$$\begin{aligned}\text{ordered-better}(\mathbf{v}, \mathbf{v}') \equiv\ & \exists k \in [1, h], \forall k' \in [1, k-1],\\ & (\forall c_{k',i} \in H_{k'}, \mathrm{error}(c_{k',i}, \mathbf{v}) = \mathrm{error}(c_{k',i}, \mathbf{v}')) \wedge\\ & (\exists c_{k,i} \in H_k, (\forall c_{k,i'} \in H_k, i' < i \Rightarrow\\ & \qquad \mathrm{error}(c_{k,i'}, \mathbf{v}) = \mathrm{error}(c_{k,i'}, \mathbf{v}')) \wedge\\ & \qquad \mathrm{error}(c_{k,i}, \mathbf{v}) < \mathrm{error}(c_{k,i}, \mathbf{v}')).\end{aligned}$$

An advantage of ordered-better is that any ordered-better solution is also a locally-better solution, which is guaranteed by the following theorem [5].

Theorem 1: Let H be an arbitrary constraint hierarchy, and $S_{\mathrm{OB}}(H)$ and $S_{\mathrm{LB}}(H)$ be its ordered-better and its locally-better solution set respectively. Then the following holds: $S_{\mathrm{OB}}(H) \subseteq S_{\mathrm{LB}}(H)$.

Versions of ordered-better are derived from different error functions. Ordered-error-better (OEB) adopts the same error function as LEB, and can be used instead of LEB.

IV. The HiRise2 Algorithm

Now we propose the HiRise2 algorithm for solving hierarchies of linear constraints.

A. Problem Statement

HiRise2 treats linear constraints $\mathbf{a}^{\mathrm{T}}\mathbf{x} \bowtie d$ with $\mathbf{a} \in \mathbb{R}^n$, $\bowtie \in \{=, \ge\}$, and $d \in \mathbb{R}$. In addition, it supports *edit* and *stay* constraints [13].

- An edit constraint is given to some variable x_j, and is internally represented as $x_j = d$. It is repeatedly updated by receiving value d from outside. Edit constraints handle inputs to variables, and are used, for example, when an object is moved by a mouse in a user interface.

- A stay constraint is associated with some variable x_j, and is internally expressed as $x_j = d$, where d is set to the value of x_j just before constraint solving. Stay constraints are typically declared to be preferential, and are used to keep variable values as long as they are satisfiable in constraint hierarchies.

HiRise2 solves hierarchies of linear constraints as ordered constraint hierarchies by using OEB. Constraints with strengths 0 to $h-1$ are given from outside. Also, for each x_j, a default stay constraint with strength h is internally generated to prevent the solution from changing unexpectedly. Therefore we can assume that the number of constraints is always no smaller than the number of variables.

B. Hierarchical Linear Programming

Transforming a given constraint hierarchy H with n variables and m constraints (where $m = \sum_{k\in[0,h]} m_k \geq n$), HiRise2 constructs the following optimization problem that we call a *hierarchical linear programming* (HLP) problem:

$$\begin{array}{l} \text{minimize } \mathbf{z} = W\mathbf{y} \\ \text{subject to } A\mathbf{x} + B\mathbf{y} = \mathbf{d},\ \mathbf{y} \geq \mathbf{0}, \end{array}$$

where $\mathbf{x}$ is an n-dimensional real vector representing *unrestricted variables* (that may take any real numbers), $\mathbf{y}$ is a $2m$-dimensional real vector representing *restricted variables* (that may take nonnegative real numbers), A is an $m \times n$ real matrix, B is an $m \times 2m$ real matrix, d is an m-dimensional real vector, z is an m-dimensional real vector, and W is an $m \times 2m$ real matrix.

The constraints $A\mathbf{x}+B\mathbf{y} = \mathbf{d}$ are obtained (in an arbitrary order) by transforming the constraints in H in the same way as Cassowary [3], [4]. Specifically, each constraint c that is in form $\mathbf{a}^{\mathrm{T}}\mathbf{x} \bowtie d$ is represented by introducing restricted variables as follows:

$$\mathbf{a}^{\mathrm{T}}\mathbf{x} = \begin{cases} d + y_{\mathrm{a}}^{+} - y_{\mathrm{a}}^{-} & \text{if } c \text{ is a required equation} \\ d + y_{\mathrm{s}} - y_{\mathrm{a}}^{-} & \text{if } c \text{ is a required inequality} \\ d + y_{\mathrm{e}}^{+} - y_{\mathrm{e}}^{-} & \text{if } c \text{ is a preferential equation} \\ d + y_{\mathrm{s}} - y_{\mathrm{e}}^{-} & \text{if } c \text{ is a preferential inequality,} \end{cases}$$

where y_{a}^{+} and y_{a}^{-} are *artificial variables*, y_{e}^{+} and y_{e}^{-} are *error variables*, y_{s} is a *slack variable*, and all these are restricted variables. After execution of HiRise2, the following happens: the artificial variables become 0 (which is not achieved when there are inconsistent required constraints, and in this case HiRise2 reports an error); error variables are made as close to 0 as possible to conform to constraint hierarchy solutions; slack variables are set to some nonnegative real numbers.

The objective function $\mathbf{z} = W\mathbf{y}$ consists of m equations $z_i = \mathbf{w}_i^{\mathrm{T}}\mathbf{y}$, where z_i is the i-th element of $\mathbf{z}$, and $\mathbf{w}_i^{\mathrm{T}}$ is the i-th row of W. Each $z_i = \mathbf{w}_i^{\mathrm{T}}\mathbf{y}$ corresponds to such $c_{k',i'}$ that $1 \leq i' \leq m_{k'}$ and $i = \sum_{k\in[0,k'-1]} m_k + i'$. In $\mathbf{w}_i^{\mathrm{T}}$, we set to 1 the element(s) corresponding to $c_{k',i'}$'s artificial/error variable(s), and set the other elements to 0. In other words, we have the following:

$$z_i = \begin{cases} y_{\mathrm{a}}^{+} + y_{\mathrm{a}}^{-} & \text{if } c_{k',i'} \text{ is a required equation} \\ y_{\mathrm{a}}^{-} & \text{if } c_{k',i'} \text{ is a required inequality} \\ y_{\mathrm{e}}^{+} + y_{\mathrm{e}}^{-} & \text{if } c_{k',i'} \text{ is a preferential equation} \\ y_{\mathrm{e}}^{-} & \text{if } c_{k',i'} \text{ is a preferential inequality,} \end{cases}$$

where y_{a}^{+}, y_{a}^{-}, y_{e}^{+}, and y_{e}^{-} are artificial and error variables for $c_{k',i'}$.

Minimizing the objective function $\mathbf{z}$ is based on the lexicographic order $<_{\mathrm{lex}}$, i.e.,

$$\mathbf{z} <_{\mathrm{lex}} \mathbf{z}' \equiv \exists i \in [1,m], (\forall i' \in [1, i-1], z_{i'} = z'_{i'}) \wedge z_i < z'_i.$$

It should be noted that, although Cassowary also uses a vector-valued objective function, it calculates a vector element for each preferential level by summing up the error variables of the constraints in the level; by contrast, HiRise2 computes a vector element for each constraint.

Solving the resulting HLP problem, we can obtain a solution to the original constraint hierarchy. The following theorem shows a stronger property that the HLP problem can be seen as being equivalent to the original hierarchy.

Theorem 2: Given any hierarchy H of linear constraints, the OEB solution set of H is equal to the set of the unrestricted variable parts of the solutions to the HLP problem constructed from H.

C. LU Decomposition Form

HiRise2 extends the LU decomposition-based simplex method [21] to solve HLP problems. In simplex methods, variables are divided into *basic* and *nonbasic* variables. Let $\mathbf{x}^*$ and $\mathbf{x}^\circ$ be the basic and the nonbasic variables of the unrestricted variables respectively, and $\mathbf{y}^*$ and $\mathbf{y}^\circ$ be the basic and the nonbasic variables of the restricted variables respectively. Although the numbers of their dimensions depend on the progress of the simplex method, we always have the following relations: $\dim(\mathbf{x}^*) + \dim(\mathbf{x}^\circ) = n$, $\dim(\mathbf{y}^*) + \dim(\mathbf{y}^\circ) = 2m$, and $\dim(\mathbf{x}^*) + \dim(\mathbf{y}^*) = m$.

We can write the HLP problem in the following way:

$$\begin{array}{l} \text{minimize } \mathbf{z} = W^*\mathbf{y}^* + W^\circ\mathbf{y}^\circ \\ \text{subject to } A^*\mathbf{x}^* + A^\circ\mathbf{x}^\circ + B^*\mathbf{y}^* + B^\circ\mathbf{y}^\circ = \mathbf{d} \\ \qquad\qquad \mathbf{y}^* \geq \mathbf{0},\ \mathbf{y}^\circ \geq \mathbf{0}. \end{array}$$

We further rewrite the first constraint equation as follows:

$$[A^*\ B^*]\begin{bmatrix} \mathbf{x}^* \\ \mathbf{y}^* \end{bmatrix} = \mathbf{d} - [A^\circ\ B^\circ]\begin{bmatrix} \mathbf{x}^\circ \\ \mathbf{y}^\circ \end{bmatrix}. \tag{1}$$

Now we introduce an LU decomposition $[A^*\ B^*] = LU$ using a lower triangular matrix L and an upper triangular matrix U whose diagonal elements are 1. Multiplying both hand sides of (1) by L^{-1} from the left, we obtain the following LU decomposition form:

$$U\begin{bmatrix} \mathbf{x}^* \\ \mathbf{y}^* \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{d}} \\ \check{\mathbf{d}} \end{bmatrix} + \begin{bmatrix} \hat{A} & \hat{B} \\ \check{A} & \check{B} \end{bmatrix}\begin{bmatrix} \mathbf{x}^\circ \\ \mathbf{y}^\circ \end{bmatrix}, \tag{2}$$

```
repeat
    find an entry variable $y^\circ_q$ or $x^\circ_q$;
    if such $y^\circ_q$ or $x^\circ_q$ was found then
        find an exit variable $y^*_p$;
        do pivoting about $(y^*_p, y^\circ_q)$ or $(y^*_p, x^\circ_q)$;
until no entry variable is found ;
```

Figure 1. Simplex optimization in HiRise2.

where $[\hat{\mathbf{d}}^{\mathrm{T}}\ \check{\mathbf{d}}^{\mathrm{T}}]^{\mathrm{T}} = L^{-1}\mathbf{d}$, $[\hat{A}^{\mathrm{T}}\ \check{A}^{\mathrm{T}}]^{\mathrm{T}} = -L^{-1}A^\circ$, and $[\hat{B}^{\mathrm{T}}\ \check{B}^{\mathrm{T}}]^{\mathrm{T}} = -L^{-1}B^\circ$.

HiRise2 achieves $L^{-1}G = U$ by computing L^{-1} and U (where we consider $G = [A^*\ B^*]$ for simplicity). In the LU decomposition-based simplex method [21], L^{-1} and U are incrementally computed when G is modified by pivoting (which is described later). For this purpose, L^{-1} is represented as the following multiplications of matrices:

$$L^{-1} = T_f T_{f-1} \cdots T_1,$$

where $T_{f'}$ for each $f' \in [1, f]$ is a transformation matrix that is either an eta matrix [21] to eliminate a column of G or a matrix to move rows of G.

The LU decomposition form has a close relation with the tableau form that is used in the ordinary simplex method. In fact, multiplying both hand sides of the LU decomposition form by U^{-1} from the left (which is easily computable), and substituting $\mathbf{y}^*$ in the objective function, we can obtain the following tableau form:

$$\begin{bmatrix} \mathbf{z} \\ \mathbf{x}^* \\ \mathbf{y}^* \end{bmatrix} = \begin{bmatrix} W^*\check{\mathbf{d}}' \\ \hat{\mathbf{d}}' \\ \check{\mathbf{d}}' \end{bmatrix} + \begin{bmatrix} W^{\mathrm{u}} & W^{\mathrm{r}} \\ \hat{A}' & \hat{B}' \\ \check{A}' & \check{B}' \end{bmatrix} \begin{bmatrix} \mathbf{x}^\circ \\ \mathbf{y}^\circ \end{bmatrix}, \quad (3)$$

where $[(\hat{\mathbf{d}}')^{\mathrm{T}}\ (\check{\mathbf{d}}')^{\mathrm{T}}]^{\mathrm{T}} = U^{-1}[\hat{\mathbf{d}}^{\mathrm{T}}\ \check{\mathbf{d}}^{\mathrm{T}}]^{\mathrm{T}}$, $[(\hat{A}')^{\mathrm{T}}\ (\check{A}')^{\mathrm{T}}]^{\mathrm{T}} = U^{-1}[\hat{A}^{\mathrm{T}}\ \check{A}^{\mathrm{T}}]^{\mathrm{T}}$, $[(\hat{B}')^{\mathrm{T}}\ (\check{B}')^{\mathrm{T}}]^{\mathrm{T}} = U^{-1}[\hat{B}^{\mathrm{T}}\ \check{B}^{\mathrm{T}}]^{\mathrm{T}}$, $W^{\mathrm{u}} = W^*\check{A}'$, and $W^{\mathrm{r}} = W^*\check{B}' + W^\circ$. As with the tableau form for the ordinary simplex method, the tentative solution is $\mathbf{x}^* = \hat{\mathbf{d}}'$, $\mathbf{y}^* = \check{\mathbf{d}}'$, $\mathbf{x}^\circ = \mathbf{0}$, and $\mathbf{y}^\circ = \mathbf{0}$.

D. Simplex Optimization

To solve an HLP problem, HiRise2 performs simplex optimization that repeatedly updates its LU decomposition form (2) by applying pivoting operations as shown in Figure 1, which is basically the same as the ordinary simplex method. A pivoting operation exchanges a basic variable called an *exit* with a nonbasic variable called an *entry*, in order to decrease the value of the objective function.

Before pivoting, the algorithm needs to find a pair of an exit and an entry variable. Unlike the ordinary simplex method, HiRise2 treats a vector-valued objective function and unrestricted variables. In this sense, HiRise2 is similar to Cassowary, but is still largely different in how to construct a vector-valued objective function.

Reconsider the tableau form (3), and let each $\mathbf{w}^{\mathrm{u}}_j$ be the j-th column of W^{u}, each $\mathbf{w}^{\mathrm{r}}_j$ be the j-th column of W^{r}, each d'_i be the i-th element of $[(\hat{\mathbf{d}}')^{\mathrm{T}}\ (\check{\mathbf{d}}')^{\mathrm{T}}]^{\mathrm{T}}$, each $a'_{i,j}$ be the (i,j)-element of $[(\hat{A}')^{\mathrm{T}}\ (\check{A}')^{\mathrm{T}}]^{\mathrm{T}}$, and each $b'_{i,j}$ be the (i,j)-element of $[(\hat{B}')^{\mathrm{T}}\ (\check{B}')^{\mathrm{T}}]^{\mathrm{T}}$. Then the algorithm finds a pair of an exit and an entry variable (y^*_p, y°_q) or (y^*_p, x°_q) as follows:

- It finds such (y^*_p, y°_q) that $\mathbf{w}^{\mathrm{r}}_q <_{\mathrm{lex}} \mathbf{0} \wedge b'_{p,q} < 0 \wedge \forall i \in [1, \dim(\mathbf{y}^*)], b'_{i,q} < 0 \Rightarrow d'_i/b'_{i,q} \leq d'_p/b'_{p,q}$;
- It finds such (y^*_p, x°_q) that $\mathbf{w}^{\mathrm{u}}_q <_{\mathrm{lex}} \mathbf{0} \wedge a'_{p,q} \neq 0 \wedge \forall i \in [1, \dim(\mathbf{y}^*)], a'_{i,q} \neq 0 \Rightarrow |d'_i/a'_{i,q}| \geq |d'_p/a'_{p,q}|$.

If there is no such pair found, the solution is optimal, and the simplex optimization terminates. It should also be noted that, when an exit variable is searched for, an entry variable is already known; this means that the algorithm can predetermine an entry variable as shown in Figure 1.

After finding the exit and entry variables, the algorithm updates the LU decomposition form by applying a pivoting operation. For pivoting about (y^*_p, y°_q), we adopt a technique called the Forrest-Tomlin method [22]. For pivoting about (y^*_p, x°_q), we use a straightforward method.

It should be noted that guaranteeing the termination of the algorithm requires similar treatments to those for the ordinary simplex method, such as cycling prevention [21].

E. Incremental Constraint Solving

HiRise2 performs incremental constraint solving for the following four operations.

1) Adding a Constraint: When a new constraint is added to the constraint hierarchy that is already solved, HiRise2 incrementally processes the added constraint without solving the hierarchy from scratch. For this purpose it needs to update the LU decomposition form by keeping the non-negativity of the restricted variables.

Consider the following LU decomposition form before the addition:

$$\begin{bmatrix} \hat{U}^{\mathrm{u}} & \hat{U}^{\mathrm{r}} \\ & \check{U} \end{bmatrix} \begin{bmatrix} \mathbf{x}^* \\ \mathbf{y}^* \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{d}} \\ \check{\mathbf{d}} \end{bmatrix} + \begin{bmatrix} \hat{A} & \hat{B} \\ \check{A} & \check{B} \end{bmatrix} \begin{bmatrix} \mathbf{x}^\circ \\ \mathbf{y}^\circ \end{bmatrix}.$$

Let $(\mathbf{a}^*)^{\mathrm{T}}\mathbf{x}^* + (\mathbf{a}^\circ)^{\mathrm{T}}\mathbf{x}^\circ = d + y^+ - y^-$ be the added constraint with its new restricted variables y^+ and y^-. Then consider the following:

$$\begin{bmatrix} \hat{U}^{\mathrm{u}} & \hat{U}^{\mathrm{r}} & \\ & \check{U} & \\ (\mathbf{a}^*)^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}^* \\ \mathbf{y}^* \\ y^- \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{d}} \\ \check{\mathbf{d}} \\ d \end{bmatrix} + \begin{bmatrix} \hat{A} & \hat{B} & \\ \check{A} & \check{B} & \\ -(\mathbf{a}^\circ)^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}^\circ \\ \mathbf{y}^\circ \\ y^+ \end{bmatrix}.$$

By multiplying both hand sides by appropriate transformation matrices, we can transform the matrix on the left-hand side into an upper triangular one. Let d' be the value obtained by transforming d after this process. If $d' \geq 0$, we have $y^- = d' \geq 0$, and therefore the result of this transformation can be used as an LU decomposition form. If $d' < 0$, we consider the alternative equation that uses y^+ as a basic variable and y^- as a nonbasic variable, by which we can obtain an LU decomposition form.

2) Removing a Constraint: HiRise2 also supports incremental removal of an existing constraint from a hierarchy. As with adding a constraint, it needs to update the LU decomposition form appropriately. Firstly it needs to have one of the restricted variables of the removed constraint as a basic variable. If both are nonbasic, one of them is turned into a basic variable by using the same technique as Cassowary [3], [4].

Let $(\mathbf{a}_i^*)^{\mathrm{T}}\mathbf{x}^* + (\mathbf{a}_i^\circ)^{\mathrm{T}}\mathbf{x}^\circ = d_i + y_q^\circ - y_p^*$ be the constraint that we remove. Then the equation (1) before the removal can be written as follows:

$$\begin{bmatrix} A_-^* & B_{-,-}^* & \mathbf{0} & B_{-,+}^* \\ (\mathbf{a}_i^*)^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & 1 & \mathbf{0}^{\mathrm{T}} \\ A_+^* & B_{+,-}^* & \mathbf{0} & B_{+,+}^* \end{bmatrix} \begin{bmatrix} \mathbf{x}^* \\ \mathbf{y}_-^* \\ y_p^* \\ \mathbf{y}_+^* \end{bmatrix} = \begin{bmatrix} \mathbf{d}_- \\ d_i \\ \mathbf{d}_+ \end{bmatrix} - \begin{bmatrix} A_-^\circ & B_{-,-}^\circ & \mathbf{0} & B_{-,+}^\circ \\ (\mathbf{a}_i^\circ)^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & 1 & \mathbf{0}^{\mathrm{T}} \\ A_+^\circ & B_{+,-}^\circ & \mathbf{0} & B_{+,+}^\circ \end{bmatrix} \begin{bmatrix} \mathbf{x}^\circ \\ \mathbf{y}_-^\circ \\ y_q^\circ \\ \mathbf{y}_+^\circ \end{bmatrix}.$$

Note that the values of $\mathbf{x}^*$, $\mathbf{y}_-^*$, and $\mathbf{y}_+^*$ in the corresponding solution do not depend on the constraint that is removed. Therefore $\mathbf{y}_-^*$ and $\mathbf{y}_+^*$ remain nonnegative in the following equation after the constraint is removed:

$$\begin{bmatrix} A_-^* & B_{-,-}^* & B_{-,+}^* \\ A_+^* & B_{+,-}^* & B_{+,+}^* \end{bmatrix} \begin{bmatrix} \mathbf{x}^* \\ \mathbf{y}_-^* \\ \mathbf{y}_+^* \end{bmatrix} = \begin{bmatrix} \mathbf{d}_- \\ \mathbf{d}_+ \end{bmatrix} - \begin{bmatrix} A_-^\circ & B_{-,-}^\circ & B_{-,+}^\circ \\ A_+^\circ & B_{+,-}^\circ & B_{+,+}^\circ \end{bmatrix} \begin{bmatrix} \mathbf{x}^\circ \\ \mathbf{y}_-^\circ \\ \mathbf{y}_+^\circ \end{bmatrix}.$$

The algorithm obtains the LU decomposition form by computing the LU decomposition of the matrix on the left-hand side from scratch. It should be noted that this process is expected to be faster than computing the LU decomposition form from scratch, since it reuses the basic and the nonbasic variable set.

3) Processing a Stay Constraint: HiRise2 incrementally processes existing stay constraints. The basic idea behind this operation is similar to that for Cassowary [3], [4] (and the algorithm is simpler than that for Cassowary). Before constraint solving, it updates vector $\mathbf{d}$ in equation (1) by changing its elements corresponding to stay constraints; if an element d_i of $\mathbf{d}$ corresponds to a stay constraint for variable x_j, it is changed into the value of x_j in the previous solution. Then the algorithm obtains a new vector $[\hat{\mathbf{d}}^{\mathrm{T}}\ \check{\mathbf{d}}^{\mathrm{T}}]^{\mathrm{T}}$ in (2) by multiplying the updated $\mathbf{d}$ by L^{-1}. It should be noted that the nonnegativity of the restricted variables is not violated by this algorithm.

4) Processing an Edit Constraint: HiRise2 incrementally processes existing edit constraints, which is also based on the same idea as Cassowary [3], [4]. In the incremental

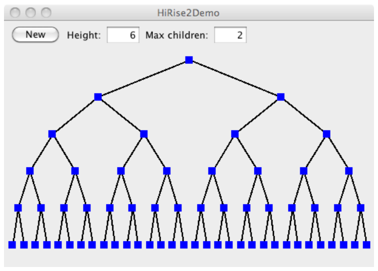
HiRise2Demo
New Height: 6 Max children: 2

Figure 2. A binary tree of height 6.

simplex method, edit constraints require more care than stay constraints, because simply changing vector $\mathbf{d}$ in equation (1) may violate the nonnegativity of the restricted variables. If such a violation happens, the algorithm fixes it by dual simplex optimization as with Cassowary.

V. Implementation

Using the HiRise2 algorithm described in the previous section, we implemented a constraint solver. It is written in the Scala language, and is usable also for Java programs. Its application programming interface is similar to those of HiRise [8]–[11] and Cassowary [3], [4]; the programmer constructs variables and constraints as objects, and adds to or removes from the constraint solver such constraint objects.

VI. Experiments

We performed experiments on the HiRise2 algorithm by using the implementation described in the previous section. For the experiments we used the constraint-based layout of binary trees as illustrated in Figure 2. We executed the experiments on a 2.3 GHz dual-core Core i5 processor with 4 GB of memory. We compared the performance of HiRise2 with that of the Java implementation of HiRise [10].

We performed two experiments. In the first experiment, we included only small numbers of inequality constraints. In this case, HiRise was quite fast as shown in Table I. In the second experiment, we included numerous inequality constraints by specifying the positional range of each node. In this case, HiRise was largely slowed down as shown in Table II. By contrast, the performance of HiRise2 was not much degraded; this was due to the integration of the simplex method into the core of its algorithm.

VII. Conclusions and Future Work

In this paper we proposed the HiRise2 constraint solving algorithm for user interface applications. It efficiently processes hierarchies of linear constraints by exploiting ordered constraint hierarchies and the LU decomposition-based simplex method. We also presented its implementation and the results of experiments on its performance.

Our future work includes showing the practical usefulness of HiRise2 by developing several user interface applications. Another future direction is to extend HiRise2 to support

Table I
EXPERIMENT ON THE PROCESSING OF SMALL NUMBERS OF INEQUALITY CONSTRAINTS.

Tree		# of constraints			# of	Exec. times (ms)	
heights	Operations	Add	Remove	Total	inequalities	HiRise2	HiRise
6	Initial layout	264	4	260	6	49	14
	Start move	2	0	262	6	3	1
	Repeat move	0	0	262	6	2	1
	Finish move	0	2	260	6	8	1
	Add node	6	2	264	6	11	1
7	Initial layout	520	4	516	6	248	64
	Start move	2	0	518	6	7	1
	Repeat move	0	0	518	6	4	1
	Finish move	0	2	516	6	20	1
	Add node	6	2	520	6	25	1
8	Initial layout	1032	4	1028	6	1632	441
	Start move	2	0	1030	6	15	2
	Repeat move	0	0	1030	6	6	1
	Finish move	0	2	1028	6	62	2
	Add node	6	2	1032	6	89	4

Table II
EXPERIMENT ON THE PROCESSING OF LARGE NUMBERS OF INEQUALITY CONSTRAINTS.

Tree		# of constraints			# of	Exec. times (ms)	
heights	Operations	Add	Remove	Total	inequalities	HiRise2	HiRise
6	Initial layout	512	4	508	254	329	53
	Start move	2	0	510	254	7	25
	Repeat move	0	0	510	254	4	1
	Finish move	0	2	508	254	23	25
	Add node	10	2	516	258	33	34
7	Initial layout	1024	4	1020	510	1980	327
	Start move	2	0	1022	510	11	156
	Repeat move	0	0	1022	510	8	1
	Finish move	0	2	1020	510	97	157
	Add node	10	2	1028	514	107	164
8	Initial layout	2048	4	2044	1022	11338	2546
	Start move	2	0	2046	1022	36	1345
	Repeat move	0	0	2046	1022	17	4
	Finish move	0	2	2044	1022	351	1349
	Add node	10	2	2052	1026	420	1359

more complex constraints such as disjunctive linear constraints [14].

REFERENCES

[1] I. E. Sutherland, "Sketchpad: A man-machine graphical communication system," in *Proc. AFIPS Spring Joint Conf.*, 1963, pp. 329–346.

[2] A. Borning, B. Freeman-Benson, and M. Wilson, "Constraint hierarchies," *Lisp Symbolic Comput.*, vol. 5, no. 3, pp. 223–270, 1992.

[3] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao, "Solving linear arithmetic constraints for user interface applications," in *Proc. ACM UIST*, 1997, pp. 87–96.

[4] G. J. Badros, A. Borning, and P. J. Stuckey, "The Cassowary linear arithmetic constraint solving algorithm," *ACM Trans. Comput.-Human Interact.*, vol. 8, no. 4, pp. 267–306, 2001.

[5] A. Wolf, "Transforming ordered constraint hierarchies into ordinary constraint systems," in *Over-Constrained Systems*, ser. LNCS, vol. 1106, 1996, pp. 171–187.

[6] H. Rudová, "Constraints with variables' annotations and constraint hierarchies," in *Proc. SOFSEM*, ser. LNCS, vol. 1521, 1998, pp. 409–418.

[7] A. Borning and G. J. Badros, "On finding graphically plausible solutions to constraint hierarchies: The split stay problem," in *CP2000 Workshop on Soft Constraints: Theory and Practice*, 2000.

[8] H. Hosobe, S. Matsuoka, and A. Yonezawa, "HiRise: An incremental constraint solver for constructing graphical user interfaces," *Comput. Softw.*, vol. 16, no. 6, pp. 33–45, 1999, in Japanese.

[9] H. Hosobe, "Speeding up HiRise, a linear constraint hierarchy solver for graphical user interfaces," *Comput. Softw.*, vol. 17, no. 2, pp. 25–29, 2000, in Japanese.

[10] ——, "A scalable linear constraint solver for user interface construction," in *Proc. CP*, ser. LNCS, vol. 1894, 2000, pp. 218–232.

[11] ——, "A linear equality and inequality constraint solver for user interfaces," *Comput. Softw.*, vol. 19, no. 6, pp. 13–20, 2002, in Japanese.

[12] B. N. Freeman-Benson, J. Maloney, and A. Borning, "An incremental constraint solver," *Comm. ACM*, vol. 33, no. 1, pp. 54–63, 1990.

[13] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning, "Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm," *Softw. Pract. Exper.*, vol. 23, no. 5, pp. 529–566, 1993.

[14] K. Marriott, P. Moulder, P. J. Stuckey, and A. Borning, "Solving disjunctive constraints for interactive graphical applications," in *Proc. CP*, ser. LNCS, vol. 2239, 2001, pp. 361–376.

[15] K. Marriott, S. S. Chok, and A. Finlay, "A tableau based constraint solving toolkit for interactive graphical applications," in *Proc. CP*, ser. LNCS, vol. 1520, 1998, pp. 340–354.

[16] K. Marriott and S. S. Chok, "QOCA: A constraint solving toolkit for interactive graphical applications," *Constraints*, vol. 7, no. 3–4, pp. 229–254, 2002.

[17] H. Hosobe, "A modular geometric constraint solver for user interface applications," in *Proc. ACM UIST*, 2001, pp. 91–100.

[18] N. Hurst, K. Marriott, and P. Moulder, "Dynamic approximation of complex graphical constraints by linear constraints," in *Proc. ACM UIST*, 2002, pp. 191–200.

[19] C. Jermann, G. Trombettoni, B. Neveu, and P. Mathis, "Decomposition of geometric constraint systems: A survey," *Intl. J. Comput. Geom. Appl.*, vol. 16, no. 5–6, pp. 379–414, 2006.

[20] A. Borning, R. Anderson, and B. Freeman-Benson, "Indigo: A local propagation algorithm for inequality constraints," in *Proc. ACM UIST*, 1996, pp. 129–136.

[21] V. Chvátal, *Linear Programming*. W. H. Freeman, 1983.

[22] J. J. H. Forrest and J. A. Tomlin, "Updated triangular factors of the basis to maintain sparsity in the product form simplex method," *Math. Prog.*, vol. 2, pp. 263–278, 1972.