The Cassowary Linear Arithmetic Constraint Solving Algorithm

Greg J. Badros, Alan Borning
Dept. of Computer Science and Engineering, University of Washington
and
Peter Stuckey
Dept. of Computer Science and Software Engineering, University of Melbourne

Linear equality and inequality constraints arise naturally in specifying many aspects of user interfaces, such as requiring that one window be to the left of another, requiring that a pane occupy the leftmost third of a window, or preferring that an object be contained within a rectangle if possible. Previous constraint solvers designed for user interface applications cannot handle simultaneous linear equations and inequalities efficiently. The inability to handle these cyclic constraints is a major limitation as they arise often in natural declarative specifications. We describe Cassowary—an incremental algorithm based on the dual simplex method that can solve such systems of constraints efficiently. We have implemented the algorithm as part of a constraint solving toolkit. We discuss the application programming interface to that toolkit and detail its implementation.

Categories and Subject Descriptors: D.1.6 [**Software**]: Logic Programming; D.2.2 [**Software**]: Tools and Techniques; G.1.3 [**Mathematics of Computing**]: Numerical Linear Algebra; G.1.6 [**Mathematics of Computing**]: Optimization

General Terms: Constraints, User Interface

Additional Key Words and Phrases: Cassowary, constraint-solving toolkit

This research has been funded in part by both a National Science Foundation Graduate Research Fellowship and the University of Washington Computer Science and Engineering Wilma Bradley fellowship for Greg Badros and in part by NSF Grant No. IIS-9975990. Alan Borning's visit to Monash University and the University of Melbourne was sponsored in part by the Australian-American Educational Foundation (Fulbright Commission).
Name: Greg Badros, Alan Borning
Address: Box 352350, Seattle, WA 98195-2350, USA {gjb,borning}@cs.washington.edu
Name: Peter Stuckey
Address: Parkville, Victoria 3052, Australia pjs@cs.mu.oz.au

1. INTRODUCTION

Linear equality and inequality constraints arise naturally in specifying many aspects of user interfaces especially layout and other geometric relations. Inequality constraints, in particular, are needed to express relationships such as "inside," "above," "below," "left-of," "right-of," and "overlaps." For example, if we are designing a Web document we can express the requirement that figure1 be to the left of figure2 as the constraint figure1.rightSide $\leq$ figure2.leftSide.

It is important to be able to express preferences as well as requirements in a constraint system used for graphical layout. For example, we must be able to express a desire for stability when moving parts of an image: things should stay where they were unless there is some reason for them to move. A second use for preferred constraints is to cope gracefully with invalid user inputs. For example, if the user tries to move a figure outside of its bounding window, it is reasonable for the figure just to bump up against the side of the window and stop, rather than causing an exception. A third use of non-required constraints is to balance conflicting desires, for example in laying out a graph.

Efficient techniques are available for solving such systems of linear constraints if the constraint network is acyclic. However, in trying to apply constraint solvers to real-world problems, we found that the collection of constraints was often cyclic. Cycles sometimes arose when the programmer unwittingly added redundant constraints—the cycles *could* have been avoided by careful analysis. However, the analysis is an added burden on the programmer. Further, it is clearly contrary to the spirit of the whole enterprise to require programmers to be constantly on guard to avoid cycles and redundant constraints; after all, one of the goals in providing constraints is to allow programmers to state what relations they want to hold in a declarative fashion, leaving it to the underlying system to enforce these relations. For other applications, such as complex layout problems with conflicting goals, cycles seem unavoidable. A solver that can handle cycles of both equality and inequality constraints is thus highly desirable.

1.1 Constraint Hierarchies and Comparators

Since we want to be able to express preferences as well as requirements in the constraint system, we need a specification for how conflicting preferences are to be traded off. *Constraint hierarchies* [Borning et al. 1992] provide a general theory for this. In a constraint hierarchy each constraint has a strength. The required strength is special, in that required constraints must be satisfied. The other strengths all label non-required constraints. A constraint of a given strength completely dominates any constraint with a weaker strength—the strong constraint must be satisfied as well as possible before the weaker constraint can have any effect on the solution. In the theory, a *comparator* is used to compare different possible solutions to the constraints and select among them.

Within this framework a number of variations are possible. One decision is whether we only compare solutions on a constraint-by-constraint basis (a *local* comparator), or whether we take some aggregate measure of the unsatisfied constraints of a given strength (a *global* comparator). A second choice is whether we are concerned only whether a constraint is satisfied or not (a *predicate* comparator), or

whether we also want to know how nearly satisfied it is (a *metric* comparator). Constraints whose domain is a metric space such as the real numbers can have an associated error function. The error in satisfying a constraint is 0 if and only if the constraint is satisfied, and becomes larger the less nearly satisfied the constraint is.

For inequality constraints it is important to use a metric rather than a predicate comparator [Borning et al. 1996]. Thus, plausible comparators for use with linear equality and inequality constraints are *locally-error-better*, *weighted-sum-better*, and *least-squares-better*. For a given collection of constraints, Cassowary finds a locally-error-better or a weighted-sum-better solution. (The related QOCA algorithm finds a least-squares-better solution which strongly penalizes outlying values when weighing constraints of the same strength [Borning et al. 1997].) The locally-error-better comparator is more permissive in that it admits more solutions to the constraints. Also, it is generally easier to develop efficient algorithms to find a locally-error-better solution because these can often be found using greedy algorithms.

1.2 Adapting the Simplex Algorithm

Linear programming is concerned with solving the following problem:

> Consider a collection of n real-valued variables $x_1, \ldots, x_n$, each of which is constrained to be non-negative: $x_i \geq 0$ for $1 \leq i \leq n$. Suppose there are m linear equality or inequality constraints over the x_i, each of the form:
>
> $a_1x_1 + \ldots + a_nx_n = b$,
> $a_1x_1 + \ldots + a_nx_n \leq b$, or
> $a_1x_1 + \ldots + a_nx_n \geq b$.
>
> Given these constraints, find values for the x_i that minimize (or maximize) the value of the *objective function*
>
> $c + d_1x_1 + \ldots + d_nx_n$.

This problem has been heavily studied for the past fifty years. The most commonly used technique for solving it is the simplex algorithm, developed by Dantzig in the 1940s, and there are now numerous variations of it. Unfortunately, existing implementations of the simplex algorithm are not readily usable for user interface applications.

The principal difficulty is incrementality. For interactive graphical applications, we need to solve similar problems repeatedly, rather than solving a single problem once. To achieve interactive response times, fast incremental algorithms that exploit prior computations are needed. There are two common cases that algorithmic changes should try to improve. First, when moving an object with a mouse or other input device, we typically represent this interaction as a one-way constraint relating the mouse position to the desired x and y coordinates of a part of the figure. For each screen refresh, we must re-satisfy the same collection of constraints while varying only the mouse location input. The second common need that incremental algorithms can optimize is when editing an object in a complex system. Ideally, when adding or removing a small number of constraints, we would like to avoid re-solving the entire system. Although the performance requirements for this case

are less stringent than for the first case, we still wish to increase performance by reusing as much of the previous solution as possible.

Another important issue when applying simplex to user interface applications is defining a suitable objective function. We must accommodate non-required constraints of different strengths which is analogous to multi-objective linear programming problems. Also, the objective function in the standard simplex algorithm must be a linear expression; but the objective functions for the locally-error-better, weighted-sum-better, and least-squares-better comparators are all non-linear. For Cassowary, we avoid the least-squares-better comparator and use a quasi-linear objective function for our weighted-sum-better comparator (see Section 2.3).

Finally, a minor issue is accommodating variables that may take on both positive and negative values, which is generally the case in user interface applications. (The standard simplex algorithm requires all variables to be non-negative.) Here we adopt efficient techniques developed for implementing constraint logic programming languages (see Section 2.1).

1.3 Overview

We present the Cassowary algorithm for incrementally solving linear equality and inequality constraints for the locally-error-better and weighted-sum-better comparators described above. In Section 2 we present our algorithm's techniques for incrementally adding and deleting constraints from a system of constraints kept in *augmented simplex form*, a type of solved form. In Section 2 we explain our procedures for incrementally solving hierarchies of constraints when an object is moved.

The Cassowary algorithm has been implemented in Smalltalk, C++, and Java as part of our Cassowary Constraint Solving Toolkit [Badros and Borning 1999]. The library performs surprisingly well, and a summary of our results is given in Section 3. The algorithm is straightforward, and a re-implementation based on this paper is reasonable, given a knowledge of the simplex algorithm.

Appendix A contains details of our implementations of the Cassowary algorithm, and Appendix B discusses some subtleties of the comparators used for optimization.

1.4 Related Work

There is a long history of using constraints in user interfaces and interactive systems, beginning with Ivan Sutherland's pioneering Sketchpad system [Sutherland 1963]. Most of the current systems use one-way constraints (e.g., [Hudson and Smith 1996; Myers 1996]), or local propagation algorithms for acyclic collections of multi-way constraints (e.g., [Sannella et al. 1993; Vander Zanden 1996]). Indigo [Borning et al. 1996] handles acyclic collections of inequality constraints, but not cycles (simultaneous equations and inequalities). User interface systems that handle simultaneous linear equations include DETAIL [Hosobe et al. 1996] and Ultraviolet [Borning and Freeman-Benson 1995]. A number of researchers (including the second author) have experimented with a straightforward use of a simplex package in a UI constraint solver, but the speed was not satisfactory for interactive use.

Baraff [Baraff 1994] describes a quadratic optimization algorithm for solving linear constraints that arise in modeling physical systems. Finally, much of the work on constraint solvers has been in the logic programming and constraint logic pro-

gramming communities. Current constraint logic programming languages such as CLP($\mathcal{R}$) [Jaffar et al. 1992] include efficient solvers for linear equalities and inequalities. (See [Marriott and Stuckey 1998] for a survey.) However, these solvers use a refinement model of computation, in which the values determined for variables are successively refined as the computation progresses, but there is no specific notion of state and change. As a result, these systems are not especially well suited for building interactive graphical applications.

An earlier paper [Borning et al. 1997] describes both the original version of Cassowary (in much less detail than this paper), and also the related QOCA algorithm. QOCA uses the same solving technique as Cassowary, but uses a least-squares-better comparator during the optimization from basic feasible solved form [Helm et al. 1992; Helm et al. 1992]. The incremental constraint deletion procedure is described in [Huynh and Marriott 1995].

2. INCREMENTAL SIMPLEX

> As you see, the subject of linear programming is surrounded by notational and terminological thickets. Both of these thorny defenses are lovingly cultivated by a coterie of stern acolytes who have devoted themselves to the field. Actually, the basic ideas of linear programming are quite simple. — *Numerical Recipes* [Press et al. 1989, page 424].

We now describe an incremental version of the simplex algorithm, adapted for our Cassowary algorithm for interactive graphical applications. In the description we use a running example, illustrated by the diagram in Figure 1.

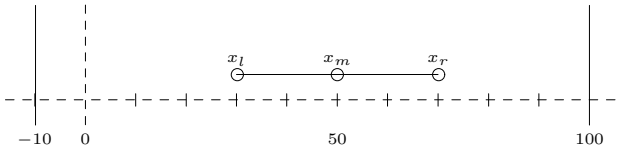

Figure 1. Simple constrained picture

The constraints on the variables in Figure 1 are as follows: x_m is constrained to be the midpoint of the line from x_l to x_r, and x_l is constrained to be at least 10 to the left of x_r. All variables must lie in the range -10 to 100. (To keep the presentation manageable, we deal only with the x coordinates. Adding analogous constraints on the y coordinates is straightforward but would double the number of constraints in our example.) Since $x_l < x_m < x_r$ in any solution, we simplify the problem by removing the redundant bounds constraints. However, even with these simplifications the resulting constraints have a cyclic constraint graph and cannot be handled by methods such as Indigo [Borning et al. 1996].

The constraints described above are

$$\begin{aligned} 2x_m &= x_l + x_r \\ x_l + 10 &\leq x_r \\ x_l &\geq -10 \\ x_r &\leq 100 \end{aligned}$$

2.1 Augmented Simplex Form

Suppose we wish to minimize the distance between x_m and x_l, or in other words, minimize $x_m - x_l$. (This simple objective function is just used as an initial example; an actual objective function is described in Section 2.3.)

The basic simplex algorithm does not itself handle variables that may take negative values (so-called *unrestricted variables*). It instead imposes an implicit constraint $x \geq 0$ on all variables occurring in its equations. Augmented simplex form allows us to handle unrestricted variables efficiently and simply; it was developed for implementing constraint logic programming languages [Marriott and Stuckey 1998], and we have adopted it here. Conceptually it uses *two* tableaux rather than one. All of the unrestricted variables from the original constraints C will be placed in C_U, the unrestricted variable tableau. C_S, the simplex tableau, contains only variables constrained to be non-negative (the *restricted variables*).

Thus, an optimization problem is in *augmented simplex form* if the constraints C have the form $C_U \wedge C_S \wedge C_I$ where C_U and C_S are conjunctions of linear arithmetic equations, C_I is $\bigwedge\{x \geq 0 \mid x \in vars(C_S)\}$, and the objective function f is a linear expression over variables in C_S.

The simplex algorithm is used to determine an optimal solution for the equations in C_S, the simplex tableau, ignoring the unrestricted variable tableau (C_U) during the optimization procedure (C_U need only be considered when finding the feasible region, prior to optimization). The equations in the C_U are then used to determine values for its unrestricted variables.

It is not difficult to re-write an arbitrary optimization problem over linear real equations and inequalities into augmented simplex form. The first step is to convert inequalities to equations. Each inequality of the form $e \leq r$, where e is a linear real expression and r is a number, can be replaced with $e + s = r \ \wedge \ s \geq 0$ where s is a newly-introduced non-negative *slack* variable. Similarly, we replace $e \geq r$ with $e - s = r \ \wedge \ s \geq 0$.

For example, the constraints for Figure 1 can be rewritten as

$$\text{minimize } x_m - x_l \quad \text{subject to}$$

$$\begin{aligned} 2x_m &= x_l + x_r \\ x_l + 10 + s_1 &= x_r \\ x_l - s_2 &= -10 \\ x_r + s_3 &= 100 \\ 0 &\leq s_1, s_2, s_3 \end{aligned}$$

We now separate the equalities into C_U and C_S. Initially all equations are in C_S. We move the unrestricted variables into C_U using Gauss-Jordan elimination. To do this, we select an equation in C_S containing an unrestricted variable u and remove the equation from C_S. We then solve the equation for u, yielding a new equation $u = e$ for some expression e. We then substitute e for all remaining occurrences of u in C_S, C_U, and f, and place the equation $u = e$ in C_U. The process is repeated until there are no more unrestricted variables in C_S. In our example, $x_r + s_3 = 100$ can be used to substitute $100 - s_3$ for x_r yielding:

minimize $x_m - x_l$ subject to

$$\begin{array}{rcll} x_r & = & 100 - s_3 & C_U \\ \hline 2x_m & = & x_l + 100 - s_3 & C_S \\ x_l + 10 + s_1 & = & 100 - s_3 & \\ x_l - s_2 & = & -10 & \\ & & & \\ 0 & \leq & s_1, s_2, s_3 & C_I \end{array}$$

Next, the first equation of C_S can be used to substitute $50 + \frac{1}{2}x_l - \frac{1}{2}s_3$ for x_m, giving

minimize $50 - \frac{1}{2}x_l - \frac{1}{2}s_3$ subject to

$$\begin{array}{rcll} x_m & = & 50 + \frac{1}{2}x_l - \frac{1}{2}s_3 & \\ x_r & = & 100 - s_3 & C_U \\ \hline x_l + 10 + s_1 & = & 100 - s_3 & C_S \\ x_l - s_2 & = & -10 & \\ & & & \\ 0 & \leq & s_1, s_2, s_3 & C_I \end{array}$$

Now we move x_l to C_U using $x_l = s_2 - 10$, giving

minimize $55 - \frac{1}{2}s_2 - \frac{1}{2}s_3$ subject to

$$\begin{array}{rcll} x_m & = & 45 + \frac{1}{2}s_2 - \frac{1}{2}s_3 & \\ x_r & = & 100 - s_3 & \\ x_l & = & s_2 - 10 & C_U \\ \hline s_2 + s_1 & = & 100 - s_3 & C_S \\ & & & \\ 0 & \leq & s_1, s_2, s_3 & C_I \end{array}$$

(Hereafter, the labels for C_U and C_S will be omitted: constraints above the horizontal line are in C_U, and constraints below the line are in C_S. Also, C_I will be omitted entirely—any variable occurring below the horizontal line is implicitly constrained to be non-negative.)

The simplex method works by taking an optimization problem in "basic feasible solved form" (a type of normal form) and repeatedly applying matrix operations to obtain new basic feasible solved forms. Once we have split the equations into C_U and C_S we can ignore C_U for purposes of optimization.

In the Cassowary implementation, all variables that may be accessed from outside the solver are unrestricted. Only error or slack variables are represented as restricted variables, and these variables occur only within the solver. (See Appendix A for further details.) The primary benefit of this simplification is that the programmer using the solver always uses just the one kind of variable. A minor benefit is that only the external, unrestricted variables actually store their values as a field in the variable object; the values of restricted variables are just given by the tableau. A minor drawback is that the constraint $v \geq 0$ must be represented explicitly. (For any other constant $c \neq 0$, $v \geq c$ must be represented explicitly in any event.)

In our running example, the constraints imply that x_r is non-negative. However, since x_r is accessible from outside the solver, we represent it as unrestricted. This does not change the solutions found. Also, we show the operations as modifying C_U as well as C_S. It would be possible to modify just C_S and leave C_U unchanged, using C_U only to define values for the variables on the left hand side of its equations. This would speed up pivoting, but it would make the incremental updates of the constants in edit constraints slower (Section 2.4). Because the latter is a much more frequent operation, we do actually modify both C_U and C_S in the implementation.

An augmented simplex form optimization problem is in *basic feasible solved form* if the equations are of the form

$$x_0 = c + a_1 x_1 + \ldots + a_n x_n$$

where the variable x_0 does not occur in any other equation or in the objective function. If the equation is in C_S, c must be non-negative. However, there is no such restriction on the constants for the equations in C_U. In either case the variable x_0 is said to be *basic* and the other variables in the equation are *parameters*. A problem in basic feasible solved form defines a *basic feasible solution*, which is obtained by setting each parametric variable to 0 and each basic variable to the value of the constant in the right-hand side.

For instance, the following constraint is in basic feasible solved form and is equivalent to the problem above.

minimize $55 - \frac{1}{2}s_2 - \frac{1}{2}s_3$ subject to

$$\begin{array}{rcl} x_l &=& -10 + s_2 \\ x_m &=& 45 + \frac{1}{2}s_2 - \frac{1}{2}s_3 \\ x_r &=& 100 - s_3 \\ \hline s_1 &=& 100 - s_2 - s_3 \end{array}$$

The basic feasible solution corresponding to this basic feasible solved form is

$$\{x_l \mapsto -10, x_m \mapsto 45, x_r \mapsto 100, s_1 \mapsto 100, s_2 \mapsto 0, s_3 \mapsto 0\}.$$

The value of the objective function with this solution is 55.

2.2 Simplex Optimization

We now describe how to find an optimum solution to a constraint in basic feasible solved form. Except for the operations on the additional unrestricted variable tableau C_U, the material presented in this subsection is simply Phase II of the standard two-phase simplex algorithm.

The simplex algorithm finds the optimum by repeatedly looking for an "adjacent" basic feasible solved form whose basic feasible solution decreases the value of the objective function that we are minimizing. When no such adjacent basic feasible solved form can be found, we have achieved an optimum. The underlying operation is called *pivoting* and involves exchanging a basic and a parametric variable using matrix operations. Thus, by "adjacent" we mean the new basic feasible solved form can be reached by performing a single pivot.

In our example, increasing s_2 from 0 will decrease the value of the objective function we are minimizing. We must be careful: we cannot increase the value of s_2 indefinitely as this may cause the value of some other basic non-negative

```
simplex_opt(C_S, f)
    repeat
        % Choose variable y_J to become basic
        if for each j ∈ {1,...,m} d_j ≥ 0 then
            return % an optimal solution has been found
        endif
        choose J ∈ {1,...,m} such that d_J < 0
        % Choose variable x_I to become non-basic
        choose I ∈ {1,...,n} such that
            −c_I/a_IJ = min_{i∈{1,...,n}}{−c_i/a_iJ | a_iJ < 0}
        e := (x_I − c_I − Σ_{j=1,j≠J}^{m} a_Ij y_j)/a_IJ
        C_S[I] := (Y_J = e)
        replace Y_J by e in f
        for each i ∈ {1,...,n}
            if i ≠ I then replace Y_J by e in C_S[I] endif
        endfor
    endrepeat
```

Figure 2. Simplex optimization

variable to become negative. We must examine the equations in C_S. The equation $s_1 = 100 - s_2 - s_3$ allows s_2 to take at most a value of 100, because if s_2 becomes larger than this, then s_1 would become negative. The equations above the horizontal line do not restrict s_2, since whatever value s_2 takes the unrestricted variables x_l and x_m can take values to satisfy the equations. In general, we choose the most restrictive equation in C_S, and use it to eliminate s_2. In the case of ties we arbitrarily break the tie. In this example, the most restrictive equation (there is only one) is $s_1 = 100-s_2-s_3$. Writing s_2 as the subject we obtain $s_2 = 100-s_1-s_3$. We replace s_2 everywhere by $100 - s_1 - s_3$ and obtain

minimize $5 + \frac{1}{2}s_1$ subject to

$$\begin{aligned} x_l &= 90 - s_1 - s_3 \\ x_m &= 95 - \tfrac{1}{2}s_1 - s_3 \\ x_r &= 100 - s_3 \\ \hline s_2 &= 100 - s_1 - s_3 \end{aligned}$$

We have just performed a pivot, having moved s_1 out of the set of basic variables and replaced it by moving s_2 into the basis.

We continue this process. Increasing the value of s_1 would increase the value of the objective function (which we are trying to minimize, so we don't want to do this). Note that decreasing s_1 would decrease the objective function's value, but as s_1 is constrained to be non-negative, it already takes its minimum value of 0 in the associated basic feasible solution. Hence we are at an optimal solution.[1]

In general, the simplex algorithm applied to C_S is described as follows. We are given a problem in basic feasible solved form in which the variables $x_1, \ldots, x_n$ are basic and the variables $y_1, \ldots, y_m$ are parameters.

[1]If we had an unrestricted variable in the objective function, the optimization would be unbounded. This possibility is not an issue for our algorithm because of the nature of the objective functions that arise from edit and stay constraints (see Section 2.3).

$$\text{minimize } e + \textstyle\sum_{j=1}^{m} d_j y_j \quad \text{subject to}$$
$$\textstyle\bigwedge_{i=1}^{n} x_i \;=\; c_i + \sum_{j=1}^{m} a_{ij} y_j \quad \wedge$$
$$\textstyle\bigwedge_{i=1}^{n} x_i \geq 0 \quad \wedge \quad \bigwedge_{j=1}^{m} y_j \geq 0.$$

Select an entry variable y_J such that $d_J < 0$. (An entry variable is one that will enter the basis, i.e., it is currently parametric and we want to make it basic.) Pivoting on such a variable can only decrease the value of the objective function. If no such variable exists, the optimum has been reached. Now determine the exit variable x_I. We must choose this variable so that it maintains basic feasible solved form by ensuring that the new c_i's are still positive after pivoting. That is, we must choose an x_I so that $-c_I/a_{IJ}$ is a minimum element of the set

$$\{-c_i/a_{iJ} \mid a_{iJ} < 0 \text{ and } 1 \leq i \leq n\}.$$

If there were no i for which $a_{iJ} < 0$ then we could stop since the optimization problem would be unbounded and so would not have a minimum: we could choose y_J to take an arbitrarily large value and thus make the objective function arbitrarily small. However, this is not an issue in our context since our optimization problems will always have a lower bound of 0.

We proceed to choose x_I, and pivot x_I out and replace it with y_J to obtain the new basic feasible solution. We continue this process until an optimum is reached. The algorithm is specified in Figure 2 and takes as inputs the simplex tableau C_S and the objective function f.

2.3 Handling Non-Required Constraints

Suppose the user wishes to edit x_m in the diagram and have x_l and x_r weakly stay where they are. This adds the non-required constraints *edit* x_m, *stay* x_l, and *stay* x_r . Suppose further that we are trying to move x_m to position 50, and that x_l and x_r are currently at 30 and 60 respectively. We are thus imposing the constraints strong $x_m = 50$, weak $x_l = 30$, and weak $x_r = 60$.

As discussed in Section 1.1, there are various possible comparators for specifying how conflicting non-required constraints are to be traded off. Cassowary finds weighted-sum-better solutions. (Since every weighted-sum-better solution is also a locally-error-better solution [Borning et al. 1992], Cassowary finds locally-error-better solutions as well.)

The error for an equality constraint $e_1 = e_2$ is defined as $|e_1 - e_2|$, while the error for an inequality constraint $e_1 \leq e_2$ is 0 if $e_1 \leq e_2$ and otherwise $e_1 - e_2$. For example, the error for the constraint $x_m = 50$ is $|x_m - 50|$.

To form an objective function for the weighted-sum-better comparator, we can sum the errors for the each constraint, weighting the errors so that satisfying any strong constraint is always strictly more important than satisfying any combination of weaker constraints. For our example, the objective function is

$$s|x_m - 50| + w|x_l - 30| + w|x_r - 60|$$

where s and w are appropriate weights. Due to the absolute value operators, this objective function is not linear, and hence the simplex method is not applicable directly. We now show how we can solve the problem using *quasi-linear optimization.*

Both the edit and the stay constraints will be represented as equations of the form

$$v = \alpha + \delta_v^+ - \delta_v^-$$

where δ_v^+ and δ_v^- are non-negative variables representing the deviation of v from the desired value α. If the constraint is satisfied both δ_v^+ and δ_v^- will be 0. Otherwise δ_v^+ will be positive and δ_v^- will be 0 if v is too big, or vice versa if v is too small.[2] Because we want δ_v^+ and δ_v^- to be 0 if possible, we make them part of the objective function, with larger coefficients for the error variables of stronger constraints. (We need to use the pair of variables to satisfy simplex's non-negativity restriction, since these variables δ_v^+ and δ_v^- will be part of the objective function.)

Translating the constraints strong $x_m = 50$, weak $x_l = 30$, and weak $x_r = 60$ which arise from the edit and stay constraints we obtain:

$$\begin{aligned} x_m &= 50 + \delta_{x_m}^+ - \delta_{x_m}^- \\ x_l &= 30 + \delta_{x_l}^+ - \delta_{x_l}^- \\ x_r &= 60 + \delta_{x_r}^+ - \delta_{x_r}^- \\ 0 &\le \delta_{x_m}^+, \delta_{x_m}^-, \delta_{x_l}^+, \delta_{x_l}^-, \delta_{x_r}^+, \delta_{x_r}^- \end{aligned}$$

as well as the original constraints:

$$\begin{aligned} 2x_m &= x_l + x_r \\ x_l + 10 &\le x_r \\ x_l &\ge -10 \\ x_r &\le 100 \end{aligned}$$

To ensure that strong constraints are always satisfied in preference to weak ones, Cassowary uses symbolic weights for the coefficients in the objective function, represented as tuples and ordered lexicographically, rather than real numbers. In the presentation that follows we will depict these symbolic weights as pairs, such as $[1, 2]$, which represents the symbolic weight consisting of the unit weight for the strong strength plus twice the unit weight for the weak strength. The objective function for our example can now be restated as:

$$\text{minimize } [1,0]\delta_{x_m}^+ + [1,0]\delta_{x_m}^- + [0,1]\delta_{x_l}^+ + [0,1]\delta_{x_l}^- + [0,1]\delta_{x_r}^+ + [0,1]\delta_{x_r}^-$$

(As an aside, if we were not using symbolic weights, and instead using real numbers as coefficients in the objective function, we might use $s = 1000$ and $w = 1$ for the strong and weak strengths. In that case the objective function would be

$$\text{minimize } 1000\delta_{x_m}^+ + 1000\delta_{x_m}^- + \delta_{x_l}^+ + \delta_{x_l}^- + \delta_{x_r}^+ + \delta_{x_r}^-.$$

While simpler, this technique has the danger that in some cases the weak constraints could overpower the strong ones, contrary to the solutions allowed by the constraint hierarchy theory. Symbolic weights avoid this danger.)

Returning to our example with symbolic weights as coefficients in the objective function, an optimal solution of this problem can be found using the simplex algorithm, and results in a tableau

[2] Although the equation may be satisfied with both δ_v^+ and δ_v^- non-zero, the simplex optimization itself forces at least one of them to be zero (see Section 2.4).

$$\text{minimize } [0,10] + [1,2]\delta^+_{x_m} + [1,-2]\delta^-_{x_m} + [0,2]\delta^-_{x_l} + [0,2]\delta^-_{x_r} \quad \text{subject to}$$

$$\begin{array}{rcrrrrr}
x_m &=& 50 & +\delta^+_{x_m} & -\delta^-_{x_m} & & \\
x_l &=& 30 & & & +\delta^+_{x_l} & -\delta^-_{x_l} \\
x_r &=& 70 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} \\
\hline
s_1 &=& 30 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -2\delta^+_{x_l} & +2\delta^-_{x_l} \\
s_3 &=& 30 & -2\delta^+_{x_m} & +2\delta^-_{x_m} & +\delta^+_{x_l} & -\delta^-_{x_l} \\
\delta^+_{x_r} &=& 10 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} \quad +\delta^-_{x_r} \\
s_2 &=& 40 & & & +\delta^+_{x_l} & -\delta^-_{x_l}
\end{array}$$

This corresponds to the solution $\{x_m \mapsto 50, x_l \mapsto 30, x_r \mapsto 70\}$ illustrated in Figure 1. Notice that the weak stay constraint on x_r is not satisfied ($\delta^+_{x_r}$ is non-zero, read directly from the second to last line of the above tableau).

2.4 Incrementality: Resolving the Optimization Problem

Now suppose the user moves the mouse (which is editing x_m) to $x = 60$. We wish to solve a new problem, with constraints strong $x_m = 60$, and weak $x_l = 30$ and weak $x_r = 70$ (so that x_l and x_r should stay where they are if possible).

There are two steps. First, we modify the tableau to reflect the new constraints we wish to solve. Second, we resolve the optimization problem for this modified tableau.

Let us first examine how to modify the tableau to reflect the new values of the stay constraints. This will not require re-optimizing the tableau, since we know that the new stay constraints are satisfied exactly. Suppose the previous stay value for variable v was α, and in the current solution v takes value β. The current tableau contains the information that

$$v = \alpha + \delta^+_v - \delta^-_v$$

and we need to modify this so that instead

$$v = \beta + \delta^+_v - \delta^-_v$$

There are two cases to consider: (a) both δ^+_v and δ^-_v are parameters, or (b) one of them is basic.

In case (a) v must take the value α in the current solution since both δ^+_v and δ^-_v take the value 0 and

$$v = \alpha + \delta^+_v - \delta^-_v$$

Hence $\beta = \alpha$ and no changes need to be made.

In case (b) assume without loss of generality that δ^+_v is basic. In the original equation representing the stay constraint, the coefficient for δ^+_v is the negative of the coefficient for δ^-_v. Since these variables occur in no other constraints, this relation between the coefficients will continue to hold as we perform pivots. In other words, δ^+_v and δ^-_v come in pairs: any equation that contains δ^+_v will also contain δ^-_v and vice versa. Since δ^+_v is assumed to be basic, it occurs exactly once in an equation with constant c, and further this equation also contains the only occurrence of δ^-_v. In the current solution

$$\{v \mapsto \beta, \delta^+_v \mapsto c, \delta^-_v \mapsto 0\}$$

and since the equation

$$v = \alpha + \delta_v^+ - \delta_v^-$$

holds, $\beta = \alpha + c$. To replace the equation

$$v = \alpha + \delta_v^+ - \delta_v^-$$

by

$$v = \beta + \delta_v^+ - \delta_v^-$$

we simply need to replace the constant c in the row for δ_v^+ by 0. Since there are no other occurrences of δ_v^+ and δ_v^- we have replaced the old equation with the new.

For our example, to update the tableau for the new values for the stay constraints on x_l and x_r we simply set the constant of the second to last equation (the equation for $\delta_{x_r}^+$) to 0. The tableau is now:

minimize $[0,0] + [1,2]\delta_{x_m}^+ + [1,-2]\delta_{x_m}^- + [0,2]\delta_{x_l}^- + [0,2]\delta_{x_r}^-$ subject to

$$\begin{array}{rcrlllll}
x_m &=& 50 & +\delta_{x_m}^+ & -\delta_{x_m}^- & & & \\
x_l &=& 30 & & & +\delta_{x_l}^+ & -\delta_{x_l}^- & \\
x_r &=& 70 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -\delta_{x_l}^+ & +\delta_{x_l}^- & \\
\hline
s_1 &=& 30 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -2\delta_{x_l}^+ & +2\delta_{x_l}^- & \\
s_3 &=& 30 & -2\delta_{x_m}^+ & +2\delta_{x_m}^- & +\delta_{x_l}^+ & -\delta_{x_l}^- & \\
\delta_{x_r}^+ &=& 0 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -\delta_{x_l}^+ & +\delta_{x_l}^- & +\delta_{x_r}^- \\
s_2 &=& 40 & & & +\delta_{x_l}^+ & -\delta_{x_l}^- &
\end{array}$$

(For completeness we update the constant part of the objective function, as well as its other terms, in our running example. However, the constant part of the objective function is irrelevant for the algorithm, and our implementations ignore it.)

Now let us consider the edit constraints. Suppose the previous edit value for v was α, and the new edit value for v is β. The current tableau contains the information that

$$v = \alpha + \delta_v^+ - \delta_v^-$$

and again we need to modify this so that instead

$$v = \beta + \delta_v^+ - \delta_v^-$$

To do so we must replace every occurrence of

$$\delta_v^+ - \delta_v^-$$

by

$$\beta - \alpha + \delta_v^+ - \delta_v^-$$

taking proper account of the coefficients of δ_v^+ and δ_v^-. (Again, remember that δ_v^+ and δ_v^- come in pairs.)

If either of δ_v^+ and δ_v^- is basic, this simply involves appropriately modifying the equation in which they are basic. Otherwise, if both are non-basic, then we need to change every equation of the form

$$x_i = c_i + a'_v\delta_v^+ - a'_v\delta_v^- + e$$

to

$$x_i = c_i + a'_v(\beta - \alpha) + a'_v\delta_v^+ - a'_v\delta_v^- + e$$

Hence modifying the tableau to reflect the new values of edit and stay constraints involves only changing the constant values in some equations. The modifications for stay constraints always result in a tableau in basic feasible solved form, since it never makes a constant become negative. In contrast the modifications for edit constraints may not.

To return to our example, suppose we pick up x_m with the mouse and move it to 60. Then we have that $\alpha = 50$ and $\beta = 60$, so we need to add 10 times the coefficient of $\delta_{x_m}^+$ to the constant part of every row. The modified tableau, after the updates for both the stays and edits, is

$$\text{minimize } [0,20] + [1,2]\delta_{x_m}^+ + [1,-2]\delta_{x_m}^- + [0,2]\delta_{x_l}^- + [0,2]\delta_{x_r}^- \quad \text{subject to}$$

$$\begin{array}{rcl}
x_m &=& 60 \quad +\delta_{x_m}^+ \quad -\delta_{x_m}^- \\
x_l &=& 30 \qquad\qquad\quad +\delta_{x_l}^+ \quad -\delta_{x_l}^- \\
x_r &=& 90 \quad +2\delta_{x_m}^+ \quad -2\delta_{x_m}^- \quad -\delta_{x_l}^+ \quad +\delta_{x_l}^- \\
\hline
s_1 &=& 50 \quad +2\delta_{x_m}^+ \quad -2\delta_{x_m}^- \quad -2\delta_{x_l}^+ \quad +2\delta_{x_l}^- \\
s_3 &=& 10 \quad -2\delta_{x_m}^+ \quad +2\delta_{x_m}^- \quad +\delta_{x_l}^+ \quad -\delta_{x_l}^- \\
\delta_{x_r}^+ &=& 20 \quad +2\delta_{x_m}^+ \quad -2\delta_{x_m}^- \quad -\delta_{x_l}^+ \quad +\delta_{x_l}^- \quad +\delta_{x_r}^- \\
s_2 &=& 40 \qquad\qquad\quad +\delta_{x_l}^+ \quad -\delta_{x_l}^-
\end{array}$$

Clearly it is feasible and already in optimal form, and so we have incrementally resolved the problem by simply modifying constants in the tableaux. The new tableaux give the solution $\{x_m \mapsto 60, x_l \mapsto 30, x_r \mapsto 90\}$. So sliding the midpoint rightwards has caused the right point to slide rightwards as well, but twice as far. The resulting diagram is shown at the top of Figure 3.

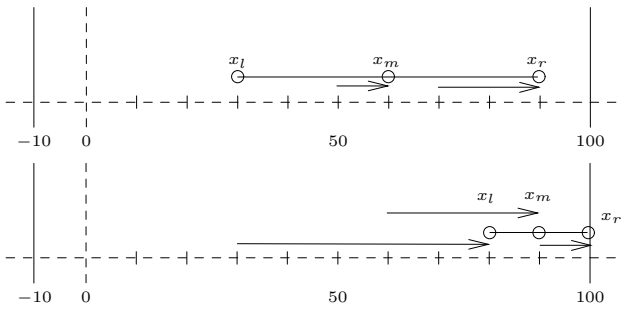

Figure 3. Resolving the constraints

Suppose we now move x_m from 60 to 90. The modified tableau is

$$\text{minimize } [0,60] + [1,2]\delta^+_{x_m} + [1,-2]\delta^-_{x_m} + [0,2]\delta^-_{x_l} + [0,2]\delta^-_{x_r} \quad \text{subject to}$$

$$\begin{array}{rcrllll}
x_m &=& 90 & +\delta^+_{x_m} & -\delta^-_{x_m} & & \\
x_l &=& 30 & & & +\delta^+_{x_l} & -\delta^-_{x_l} \\
x_r &=& 150 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} \\
\hline
s_1 &=& 110 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -2\delta^+_{x_l} & +2\delta^-_{x_l} \\
s_3 &=& -50 & -2\delta^+_{x_m} & +2\delta^-_{x_m} & +\delta^+_{x_l} & -\delta^-_{x_l} \\
\delta^+_{x_r} &=& 60 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} \quad +\delta^-_{x_r} \\
s_2 &=& 40 & & & +\delta^+_{x_l} & -\delta^-_{x_l}
\end{array}$$

The tableau is no longer in basic feasible solved form, since the constant of the row for s_3 is negative, even though s_3 is supposed to be non-negative. (In this solution $x_r = 150$, so that the right endpoint has crashed through the $x_r \leq 100$ barrier.)

Thus, in general, after updating the constants for the edit constraints, the simplex tableau C_S may no longer be in basic feasible solved form, since some of the constants may be negative. However, the tableau is still in basic form, so we can still read a solution directly from it as before. Also, because no coefficient has changed (in particular the optimization function is the same), the resulting tableau reflects an optimal but not feasible solution.

We need to find a feasible and optimal solution. We could do so by adding artificial variables (as when adding a constraint—see Section 2.5), optimizing the sum of the artificial variables to find an initial feasible solution, and then re-optimizing the original problem.

But we can do much better. The process of moving from an optimal and *infeasible* solution to an optimal and *feasible* solution is exactly the dual of normal simplex algorithm, where we progress from a feasible and non-optimal solution to feasible and optimal solution. Hence we can use the *dual simplex algorithm* to find a feasible solution while staying optimal.

Solving the dual optimization problem starts from an infeasible optimal tableau of the form

$$\text{minimize } e + \Sigma_{j=1}^m d_j y_j \quad \text{subject to}$$

$$\textstyle\bigwedge_{i=1}^n x_i = c_i + \Sigma_{j=i}^m a_{ij} y_j$$

where some c_i may be negative for rows with non-negative basic variables (accounting for the tableau's infeasibility) and each d_j is non-negative (so it is optimal).

The dual simplex algorithm selects an exit variable by finding a row I with non-negative basic variable x_I and negative constant c_I. The entry variable is the variable y_J such that the ratio d_J/a_{IJ} is the minimum of all d_j/a_{Ij} where a_{Ij} is positive. This selection criteria ensures that when pivoting we stay at an optimal solution. The pivot replaces y_j by

$$-1/a_{Ij}(-x_I + c_I + \Sigma_{j=1, j\neq J}^m a_{Ij} y_j)$$

and is performed as in the (primal) simplex algorithm. The algorithm is shown in Figure 4.

Continuing the example above, we select the exit variable s_3—the only non-negative basic variable for a row with negative constant. We find that $\delta^+_{x_l}$ has the minimum ratio since its coefficient in the optimization function is 0, so it will be

```
re_optimize(C_S,f)
    foreach stay : v ∈ C
        if δ_v^+ or δ_v^- is basic in row i then c_i := 0 endif
    endfor
    foreach edit : v ∈ C
        let α and β be the previous and current edit values for v
        let δ_v^+ be y_j
        foreach i ∈ {1,...,n}
            c_i := c_i + a_ij(β − α)
        endfor
    endfor
    repeat
        % Choose variable x_I to become non-basic
        choose I where c_I < 0
        if there is no such I
            return true
        endif
        % Choose variable y_J to become basic
        if for each j ∈ {1,...,m} a_Ij ≤ 0 then
            return false
        endif
        choose J ∈ {1,...,m} such that
            d_J/a_IJ = min_{j∈{1,...,m}}{d_j/a_Ij | a_Ij > 0}
        e := (x_I − c_I − Σ_{j=1,j≠J}^{m} a_Ij y_j)/a_IJ
        replace y_J by e in f
        for each i ∈ {1,...,n}
            if i ≠ I then replace y_J by e in row i endif
        endfor
        replace the I^th row by y_J = e
    until false
```

Figure 4. Dual Simplex Re-optimization

the entry variable. Replacing $\delta^+_{x_l}$ everywhere by $50 + s_3 + 2\delta^+_{x_m} - 2\delta^-_{x_m} + \delta^-_{x_l}$ we obtain the tableau

$$\text{minimize } [0,60] + [1,2]\delta^+_{x_m} + [1,-2]\delta^-_{x_m} + [0,2]\delta^-_{x_l} + [0,2]\delta^-_{x_r} \quad \text{subject to}$$

$$\begin{array}{rcrrrrr}
x_m &=& 90 & & +\delta^+_{x_m} & -\delta^-_{x_m} & \\
x_l &=& 80 & +s_3 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & \\
x_r &=& 100 & -s_3 & & & \\
\hline
s_1 &=& 10 & -2s_3 & -2\delta^+_{x_m} & +2\delta^-_{x_m} & \\
\delta^+_{x_l} &=& 50 & +s_3 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & +\delta^-_{x_l} \\
\delta^+_{x_r} &=& 10 & -s_3 & & & +\delta^-_{x_r} \\
s_2 &=& 90 & +s_3 & +2\delta^+_{x_m} & -2\delta^-_{x_m} &
\end{array}$$

The tableau is feasible (and of course still optimal) and represents the solution $\{x_m \mapsto 90, x_r \mapsto 100, x_l \mapsto 80\}$. So by sliding the midpoint further right, the rightmost point hits the wall and the left point slides right to satisfy the constraints. The resulting diagram is shown at the bottom of Figure 3.

To summarize, incrementally finding a new solution for new input variables involves updating the constants in the tableaux to reflect the updated stay con-

straints, then updating the constants to reflect the updated edit constraints, and finally re-optimizing if needed. In an interactive graphical application, this dual optimization method typically requires a pivot only when one part of the figure first hits or first moves away from a barrier. The intuition behind this is that when a constraint first becomes unsatisfied, the value of one of its error variables will become non-zero, and hence the variable will have to enter the basis; conversely, when a constraint first becomes satisfied, we can move one of its error variables out of the basis.

In the example, pivoting occurred when the right point x_r came up against a barrier. Thus, if we picked up the midpoint x_m with the mouse and smoothly slid it rightwards, 1 pixel every screen refresh, only one pivot would be required in moving from 50 to 95. This behaviour is why the dual optimization is well suited to this problem and leads to efficient resolving of the hierarchical constraints.

2.5 Incrementality: Adding a Constraint

We now describe how to add the equation for a new constraint incrementally. This technique is also used in our implementation to find an initial basic feasible solved form for the original simplex problem, by starting from an empty constraint set and adding the constraints one at a time.

As an example, suppose we wish to require that the midpoint be centered, i.e. to add a required constraint $x_m = 50$ to the final tableau given in Section 2.2, page 9. For reference, that tableau is:

$$\begin{array}{rcl} x_l &=& 90 - s_1 - s_3 \\ x_m &=& 95 - \frac{1}{2}s_1 - s_3 \\ x_r &=& 100 - s_3 \\ \hline s_2 &=& 100 - s_1 - s_3 \end{array}$$

If we substitute for each of the basic variables in $x_m = 50$ (namely x_m), we obtain the equation $45 - \frac{1}{2}s_1 - s_3 = 0$. In order to add this constraint straightforwardly to the tableau we create a new non-negative variable a called an *artificial variable*. (This technique is simply an incremental version of the operation used in Phase I of the two-phase simplex algorithm.) We let $a = 45 - \frac{1}{2}s_1 - s_3$ be added to the tableau (clearly this gives a tableau in basic feasible solved form) and then minimize the value of a. If a takes the value 0 then we have obtained a solution to the problem with the added constraint, and we can then eliminate the artificial variable altogether since it is a parameter (and hence takes the value 0). This is the case for our example; the resulting tableau is

$$\begin{array}{rcl} x_l &=& 0 + s_3 \\ x_m &=& 50 \\ x_r &=& 100 - s_3 \\ \hline s_1 &=& 90 - 2s_3 \\ s_2 &=& 10 + s_3 \end{array}$$

In general, to add a new required constraint to the tableau we first convert it to an augmented simplex form equation by adding slack variables if it is an inequality. Next, we use the current tableau to substitute out all the basic variables. This gives an equation $e = c$ where e is a linear expression. If c is negative, we multiply both

sides by -1 so that the constant becomes non-negative. If e contains an unrestricted variable we use it to substitute for that variable and add the equation to the tableau above the line (i.e., to C_U). Otherwise we create a restricted artificial variable a, add the equation $a = c - e$ to the tableau below the line (i.e., to C_S), and minimize $c - e$. If the resulting minimum is not zero then the constraints are unsatisfiable. Otherwise a is either parametric or basic. If a is parametric, the column for it can be simply removed from the tableau. If it is basic, the row must have constant 0 (since we were able to achieve a value of 0 for our objective function, which is equal to a). If the row is just $a = 0$, it can be removed. Otherwise, $a = 0 + bx + e$ where $b \neq 0$. We can then pivot x into the basis using this row and remove the column for a.

If the equation being added contains any unrestricted variables after substituting out all the basic variables, as described above we do not need to use an artificial variable. Not only that, we *could not* use an artificial variable, since we cannot put an equation in C_S that contains an unrestricted variable. In some other cases we can avoid using an artificial variable for efficiency, even though it would be permissible to use one. We can avoid using an artificial variable if we can choose a subject for the equation from among its current variables. Here are the rules for choosing a subject. (These are to be used after replacing any basic variables with their defining expressions.)

We start with an expression e. If necessary, normalize e by multiplying by -1 so that its constant part is non-negative. We are adding the constraint $e = 0$ to the tableau. To do this, we want to pick a variable in e to be the subject of an equation, so that we can add the row $v = e'$, where e' the result of solving $e = 0$ for v.

—If e contains any unrestricted variables, we must choose an unrestricted variable as the subject.

—If the subject is new to the solver, we will not have to do any substitutions, so we prefer new variables to ones that are currently noted as parametric.

—If e contains only restricted variables, if there is a (restricted) variable in e that has a negative coefficient and that is new to the solver, we can pick that variable as the subject.

—Otherwise use an artificial variable.

A consequence of these rules is that we can always add a non-required constraint to the tableau without using an artificial variable, because the equation will contain a positive and a negative error or slack variable, both of which are new to the solver, and which occur with opposite signs. (Constraints that are originally equations will have a positive and a negative error variable, while constraints that are originally inequalities will have one error variable and one slack variable, with opposite signs.) This observation is good news for performance, since adding a non-required edit constraint is a common operation.

2.6 Incrementality: Removing a Constraint

We also want a method for incrementally removing a constraint from the tableaux. After a series of pivots have been performed, the information represented by the

constraint may not be contained in a single row, so we need a way to identify the constraint's influence in the tableaux. To do this, we use a "marker" variable that is originally present only in the equation representing the constraint. We can then identify the constraint's effect on the tableaux by looking for occurrences of that marker variable. For inequality constraints, the slack variable s that we added to make it an equality serves as the marker (because s will originally occur only in that equation). For non-required equality constraints, either of its two error variables can serve as a marker—see Section 2.3. For required equality constraints, we add a "dummy" restricted variable to the original equation to serve as a marker, which we never allow to enter the basis (so that it always has value 0). In our running example, then, to allow the constraint $2x_m = x_l + x_r$ to be deleted incrementally we would have added a dummy variable d, resulting in $2x_m = x_l + x_r + d$. The simplex optimization routine checks for these dummy variables in choosing an entry variable and does not allow one to be selected. These dummy variables must be restricted, not unrestricted, because we might need to have some of them in the equations for restricted basic variables. (We did not include the variable d in the tableaux presented earlier to simplify the presentation.)

Consider removing the constraint that x_l is 10 to the left of x_r. The slack variable s_1, which we added to the inequality to make it an equation, records exactly how this equation has been used to modify the tableau. We can remove the inequality by pivoting the tableau until s_1 is basic and then simply drop the row in which it is basic.

In the tableau in Section 2.5 (obtained after adding the required constraint $x_m = 50$), s_1 is already basic, so removing it simply means dropping the row in which it is basic, obtaining

$$\begin{array}{rcrl} x_l &=& 0 & +s_3 \\ x_m &=& 50 & \\ x_r &=& 100 & -s_3 \\ \hline s_2 &=& 10 & +s_3 \end{array}$$

If we wanted to remove this constraint from the tableau before adding $x_m = 50$ (i.e., the final tableau given in Section 2.2, page 9), s_1 is a parameter. We make s_1 basic by treating it as an entry variable and (as usual) determining the most restrictive equation, then using that to pivot s_1 into the basis before finally removing the row.

There is such a restrictive equation in this example. However, if the marker variable does not occur in C_S, or if its coefficients in C_S are all non-negative, then no equation restricts the value of the marker variable. If the marker variable does occur in one or more equations in C_S, always with a positive coefficient, pick the equation with the smallest ratio of the constant to the marker variable's coefficient. (The row with the marker variable will become infeasible after the pivot, but all the other rows will still be feasible, and we will be dropping the row with the marker variable. In effect we are removing the non-negativity restriction on the marker variable.) Finally, if the marker variable occurs only in equations for unrestricted variables, we can choose any equation in which it occurs.

In the final tableau in Section 2.2, page 9, the row $s_2 = 100 - s_1 - s_3$ is the most constraining equation. Pivoting to let s_1 enter the basis and then removing the row

in which it is basic, we obtain

$$\begin{array}{rcl} x_l & = & -10 + s_2 \\ x_m & = & 45 + \frac{1}{2}s_2 - \frac{1}{2}s_3 \\ x_r & = & 100 - s_3 \end{array}$$

In the preceding example the marker variable had a negative coefficient. Here is an example that only has positive coefficients. The original constraints are:

$$\begin{array}{rcl} x & \geq & 10 \\ x & \geq & 20 \\ x & \geq & 30 \end{array}$$

In basic feasible solved form, this is:

$$\begin{array}{rcl} x & = & 30 + s_3 \\ \hline s_1 & = & 20 + s_3 \\ s_2 & = & 10 + s_3 \end{array}$$

where s_1, s_2, and s_3 are the marker (and slack) variables for $x \geq 10$, $x \geq 20$, and $x \geq 30$ respectively. This gives a solution for x of $x = 30$, which of course satisfies all of the original inequalities.

Suppose we want to remove the $x \geq 30$ constraint. We need to pivot to make s_3 basic. The equation that gives the smallest ratio is $s_2 = 10 + s_3$, so the entry variable is s_3 and the exit variable is s_2, giving:

$$\begin{array}{rcl} x & = & 20 + s_2 \\ \hline s_1 & = & 10 + s_2 \\ s_3 & = & -10 + s_2 \end{array}$$

This tableau is now infeasible, but we drop the row with s_3 giving

$$\begin{array}{rcl} x & = & 20 + s_2 \\ \hline s_1 & = & 10 + s_2 \end{array}$$

which is of course feasible.

A beneficial result of using marker variables is that redundant constraints can be represented and manipulated. Consider:

$$\begin{array}{rcl} x & \geq & 10 \\ x & \geq & 10 \end{array}$$

When converted to basic feasible solved form, each $x \geq 10$ constraint gets a separate slack variable, which is used as the marker variable for that constraint.

$$\begin{array}{rcl} x & = & 10 + s_1 \\ \hline s_2 & = & 0 + s_1 \end{array}$$

To delete the second $x \geq 10$ constraint we would simply drop the $s_2 = 0 + s_1$ row. To delete the first $x \geq 10$ constraint we would pivot, making s_1 basic and s_2 parametric:

$$\begin{array}{rcl} x & = & 10 + s_2 \\ \hline s_1 & = & 0 + s_2 \end{array}$$

and then drop the $s_1 = 0 + s_2$ row.

A consequence of directly representing redundant constraints is that they must all be removed to eliminate their effect. (This seems to be a more desirable behavior for the solver than removing redundant constraints automatically, although if the latter were desired the solver could be modified to do this.) Another consequence is that when adding a new constraint, we would never decide that it was redundant and not add it to the tableau.

Before we remove the constraint, there may be some stay constraints that were unsatisfied previously. If we just removed the constraint these could come into play, so instead, we reset all of the stays so that all variables are constrained to stay at their current values.

Also, if the constraint being removed is not required we need to remove the error variables for it from the objective function. To do this we add the following to the expression for the objective function:

$$-1 \times e \times s \times w$$

where e is the error variable if it is parametric, or else e is its defining expression if it is basic, s is the unit symbolic weight for the constraint's strength, and w is its weight at the given strength. (In the implementation s is an instance of ClSymbolicWeight and w is a float—see Section A.2.5 of the Appendix.)

If we allow non-required constraints other than stays and edits, we also need to re-optimize after deleting a constraint, since a non-required constraint might have become satisfiable (or more nearly satisfiable).

3. EMPIRICAL EVALUATION

Cassowary has been implemented in Smalltalk, C++, and Java. We ran some simple benchmarks using test problems which tried to add 300 randomly-generated constraints using 300 variables, and 900 randomly-generated constraints using 900 variables.

With the Smalltalk implementation of Cassowary on the 300-constraint benchmark problem, adding a constraint takes on average 38 msec (including the initial solve), deleting a constraint 46 msec, and resolving as the point moves 15 msec. (Stay and edit constraints are represented explicitly in this implementation, so there were also stay constraints on each variable, plus two edit constraints, for a total of 602 constraints minus the constraints that, if added, would have resulted in an unsatisfiable system.) For the 900 constraint problem, adding a constraint takes on average 98 msec, deleting a constraint 151 msec, and resolving as the point moves 45 msec. These tests were run using an implementation in OTI Smalltalk Version 4.0 running on a IBM Thinkpad 760EL laptop computer.

For the C++ implementation on the problem with 900 constraints and variables, adding a constraint takes 15 msec, deleting a constraint 1.2 msec, and resolving as the point moves 1.4 msec. These tests were run on a Pentium III/450 running Linux 2.2.5 and compiled with GCC-2.95.2. The Java implementation under the

basic Sun JDK 1.2 (no JIT compiler) is about 3 to 8 times slower than the C++ implementation.

The various implementations of Cassowary are actively being used. A Scheme wrapping of the C++ implementation is used in our Scheme Constraints Window Manager (SCWM) [Badros and Stachowiak 1999; Badros et al. 2000]. We have also embedded the C++ implementation in a prototype web browser that supports our constraint-based extension to Cascading Style Sheets [Badros et al. 1999]. A demonstration Constraint Drawing Application using the Java implementation was written by Michael Noth and is included with the Cassowary toolkit. A third Cassowary application (developed using a different Java implementation) is a web authoring tool [Borning et al. 1997] in which the appearance of a page is determined by the combination of constraints from both the web author and the viewer. Cassowary has also been used in a non-interactive application to perform consistency checks in a planning application [Wolfman and Weld 1998].

4. CONCLUSION

The Cassowary algorithm is the first constraint solver for interactive user interface algorithms that handles simultaneous linear equations and inequalities. Because of the minimal update of the tableau which is performed, it is is (perhaps surprisingly) fast on the operation of incrementally resolving the system. That operation's efficiency is crucial for interactive redrawing diagrams during editing.

Additionally, because Cassowary handles cycles in the constraint graph without difficulty, users of the Cassowary implementations can concentrate on exploiting the additional expressiveness that the library provides; the declarative nature of constraints is not undermined by a need to understand the algorithm. Cassowarry has proven to be efficient and expressive enough to be used in many applications.

ACKNOWLEDGMENTS

Thanks to Kim Marriott for his work on the closely-related QOCA algorithm. We received much useful feedback from early users of our toolkit including Anthony Beurivé, Alexandre Duret-Lutz, Michael Kaufmann, Brian Grant, Pengling He, Tessa Lau, Sorin Lerner, John MacPhail, Larry Melia, Michael Noth, Emmanual Pietriga, Stefan Saroiu, and Steve Wolfman.

APPENDIX

A. IMPLEMENTATION DETAILS

A.1 Principal Classes

The principal classes in our implementations are as follows. All the classes start with "Cl" for "Constraint Library" and are, of course, direct or indirect subclasses of Object in the Smalltalk and Java implementations.

- Object
  - ClAbstractVariable
    - ClDummyVariable
    - ClObjectiveVariable
    - ClSlackVariable

- ClVariable
- ClConstraint
  - ClEditOrStayConstraint
    - ClEditConstraint
    - ClStayConstraint
  - ClLinearConstraint
    - ClLinearEqualityConstraint
    - ClLinearInequalityConstraint
- ClLinearExpression
- ClTableau
  - ClSimplexSolver
- ClStrength
- ClSymbolicWeight

Some of these classes make use of the *Dictionary* (or *map*) abstract data type: dictionaries have keys and values and permit efficiently finding the value for a given key, and adding or deleting key/value pairs. One can also iterate through all keys, all values, or all key/value pairs.

A.2 Solver Protocol

The solver itself is represented as an instance of ClSimplexSolver. Its public message protocol is as follows.

addConstraint(ClConstraint cn)
: Incrementally add the linear constraint cn to the tableau. The constraint object contains its strength.

removeConstraint(ClConstraint cn)
: Remove the constraint cn from the tableau. Also remove any error variables associated with cn from the objective function.

addEditVar(ClVariable v, ClStrength s)
: Add an edit constraint of strength s on variable v to the tableau so that suggestValue (see below) can be used on that variable after a beginEdit().

removeEditVar(ClVariable v)
: Remove the previously added edit constraint on variable v. The endEdit message automatically removes all the edit variables as part of terminating an edit manipulation.

beginEdit()
: Prepare the tableau for new values to be given to the currently-edited variables. The addEditVar message should be used before calling beginEdit, and suggestValue and resolve should be used only after beginEdit has been invoked, but before the required matching endEdit.

suggestValue(ClVariable v, double n)
: Specify a new desired value n for the variable v. Before this call, v needs to have been added as a variable of an edit constraint (either by addConstraint of a hand-built EditConstraint object or more simply using addEditVar).

endEdit()
: Denote the end of an edit manipulation, thus removing all edit constraints from

the tableau. Each beginEdit call must be matched with a corresponding endEdit invocation, and the calls may be nested properly.

resolve()
Try to re-solve the tableau given the newly specified desired values. Calls to resolve should be sandwiched between a beginEdit() and an endEdit(), and should occur after new values for edit variables are set using suggestValue.

addPointStays(Vector points)
This method is a bit of a kludge, and addresses the desire to satisfy the stays on both the x and y components of a given point rather than on the x component of one point and the y component of another. The argument points is an array of points, whose x and y components are constrainable variables. This method adds a weak stay constraint to the x and y variables of each point. The weights for the x and y components of a given point are the same. However, the weights for successive points are each smaller than those for the previous point (1/2 of the previous weight). The effect of this is to encourage the solver to satisfy the stays on both the x and y of a given point rather than the x stay on one point and the y stay on another. See Appendix B for more discussion of this issue.

setAutoSolve(boolean f)
Choose whether the solver should automatically optimize and set external variable values after each addConstraint or removeConstraint. By default, auto-solving is on, but passing false to this method will turn it off (until later turned back on by passing true to this method). When auto-solving is off, solve (below) or resolve must be invoked to see changes to the ClVariables contained in the tableau.

isAutoSolving() **returns** boolean
Return true if and only if the solver is auto-solving, false otherwise.

solve()
Optimize the tableau and set the external ClVariables contained in the tableau to their new values. This method need only be invoked if auto-solving has been turned off. It never needs to be called after a resolve method invocation.

reset()
Re-initialize the solver from the original constraints, thus getting rid of any accumulated numerical problems. (It is not yet clear how often such problems arise, but we provide the method just in case.)

A.2.1 *Variables.* ClAbstractVariable and its subclasses represent various kinds of constrained variables. ClAbstractVariable is an abstract class, that is, it is just used as a superclass of other classes; one does not make instances of ClAbstractVariable itself. ClAbstractVariable defines the message protocol for constrainable variables. Its only instance variable is name, which is a string name for the variable. (This field is used for debugging and constraint understanding tasks.)

Instances of the concrete ClVariable subclass of ClAbstractVariable are what the user of the solver sees (hence it was given a nicer class name). This class has an instance variable value that holds the value of this variable. Users of the solver can send one of these variables the message value to get its value.

The other subclasses of ClAbstractVariable are used only within the solver. They

do not hold their own values—rather, the value is just given by the current tableau. None of them have any additional instance variables.

Instances of ClSlackVariable are restricted to be non-negative. They are used as the slack variable when converting an inequality constraint to an equation and for the error variables to represent non-required constraints.

Instances of ClDummyVariable is used as a marker variable to allow required equality constraints to be deleted. (For inequalities or non-required constraints, the slack or error variable is used as the marker.) These dummy variables are never pivoted into the basis.

An instance of ClObjectiveVariable is used to index the objective row in the tableau. (Conventionally this variable is named z.) This kind of variable is just for convenience—the tableau is represented as a dictionary (with some additional cross-references). Each row is represented as an entry in the dictionary; the key is a basic variable and the value is an expression. So an instance of ClObjectiveVariable is the key for the objective row. The objective row is unique in that the coefficients of its expression are ClSymbolicWeights in the Smalltalk implementation, not just ordinary real numbers. However, the C++ and Java implementations convert ClSymbolicWeights to real numbers to avoid dealing with ClLinearExpressions parameterized on the type of the coefficient—see Section A.2.5 for more details.

All variables understand the following messages: isDummy, isExternal, isPivotable, and isRestricted. They also understand messages to get and set the variable's name.

Class	isDummy	isExternal	isPivotable	isRestricted
ClDummyVariable	true	false	false	true
ClVariable	false	true	false	false
ClSlackVariable	false	false	true	true
ClObjectiveVariable	false	false	false	false

Figure 5. Subclasses of ClAbstractVariable

For isDummy, instances of ClDummyVariable return true and others return false. The solver uses this message to test for dummy variables. It will not choose a dummy variable as the subject for a new equation, unless all the variables in the equation are dummy variables. (The solver also will not pivot on dummy variables, but this is handled by the isPivotable message.)

For isExternal, instances of ClVariable return true and others return false. If a variable responds true to this message, it means that it is known outside the solver, and so the solver needs to give it a value after solving is complete.

For isPivotable, instances of ClSlackVariable return true and others return false. The solver uses this message to decide whether it can pivot on a variable.

For isRestricted, instances of ClSlackVariable and of ClDummyVariable return true, and instances of ClVariable and ClObjectiveVariable return false. Returning true means that this variable is restricted to being non-negative.

A variable's significance is largely just its identity (as mentioned above, variables have little state: a name for debugging and a value for instances of ClVariable). The only other messages that variables understand are some messages to ClVariable for creating constraints—see Section A.2.4.

A.2.2 *Linear Expressions.* Instances of the class ClLinearExpression hold a linear expression and are used in building and representing constraints and in representing the tableau. A linear expression holds a dictionary of variables and coefficients (the keys are variables and the values are the corresponding coefficients). Only variables with non-zero coefficients are included in the dictionary; if a variable is not in this dictionary its coefficient is assumed to be zero. The other instance variable is a constant. So to represent the linear expression $a_1x_1 + \cdots + a_nx_n + c$, the dictionary would hold the key x_1 with value a_1, etc., and the constant c.

Linear expressions understand a large number of messages. Some of these are for constraint creation (see Section A.2.4). The others are to substitute an expression for a variable in the constraint, to add an expression, to find the coefficient for a variable, and so forth.

A.2.3 *Constraints.* There is an abstract class ClConstraint that serves as the superclass for other concrete classes. It defines two instance variables: strength and weight. The variable strength is the strength of this constraint in the constraint hierarchy (and should be an instance of ClStrength), while weight is a float indicating the actual weight of the constraint at its indicated strength, or nil/null if it does not have a weight. (Weights are only relevant for weighted-sum-better comparators, not for locally-error-better ones.)

Constraints understand various messages that return true or false regarding some aspect of the constraint, such as isRequired, isEditConstraint, isStayConstraint, and isInequality.

ClLinearConstraint is an abstract subclass of ClConstraint. It adds an instance variable expression, which holds an instance of ClLinearExpression. It has two concrete subclasses. An instance of ClLinearEquation represents the linear equality constraint

$$\mathsf{expression} = 0.$$

An instance of ClLinearInequality represents the constraint

$$\mathsf{expression} \geq 0.$$

The other part of the constraint class hierarchy is for edit and stay constraints (both of which are represented explicitly). ClEditOrStayConstraint has an instance field variable, which is the ClVariable with the edit or stay. Otherwise all that the two concrete subclasses do is respond appropriately to the messages isEditConstraint and isStayConstraint.

This constraint hierarchy is also intended to allow extension to include local propagation constraints (which would be another subclass of ClConstraint)—otherwise we could have made everything be a linear constraint, and eliminated the abstract class ClConstraint entirely.

A.2.4 *Constraint Creation.* This subsection describes a mechanism to allow constraints to be defined easily by programmers. The convenience afforded by our toolkit varies among languages. Smalltalk's dynamic nature makes it the most expressive. C++'s operator overloading still permits using natural infix notation. Java, however, requires using ordinary methods, and leaves us with the single option of prefix expressions when building constraints.

In Smalltalk, the messages +, -, *, and / are defined for ClVariable and ClLinearExpression to allow convenient creation of constraints by programmers. Also, ClVariable and ClLinearExpression, as well as Number, define cnEqual:, cnGEQ:, and cnLEQ: to return linear equality or inequality constraints. Thus, the Smalltalk expression

```
3*x+5 cnLEQ: y
```

returns an instance of ClLinearEquality representing the constraint $3x + 5 \leq y$. The expression is evaluated as follows: the number 3 gets the message * x. Since x is not a number, 3 sends the message * 3 to x. x is an instance of ClVariable, which understands * to return a new linear expression with a single term, namely itself times the argument. (If the argument is not a number it raises an exception that the expression is non-linear.) The linear expression representing $3x$ gets the message + with the argument 5, and returns a new linear expression representing $3x + 5$. This linear expression gets the message cnLEQ: with the argument y. It computes a new linear expression representing $y - 3x - 5$, and then returns an instance of ClLinearInequality with this expression.

(It is tempting to make this nicer by using the =, <=, and >= messages, so that one could write

```
3*x+5 <= y
```

instead but because the rest of Smalltalk expects =, <=, and >= to perform a test and return a boolean, rather than to return a constraint, this would not be a good idea.)

Similarly, in C++ the arithmetic operators are overloaded to build ClLinearExpressions from ClVariables and other ClLinearExpressions. Actual constraints are built using various constructors for ClLinearEquation or ClLinearInequality. An enumeration defines the symbolic constants cnLEQ and cnGEQ to approximate the Smalltalk interface. For example:

```
ClLinearInequality cn(3*x+5, cnLEQ, y);        // C++
```

build the constraint cn representing $3x + 5 \leq y$. In Java, the same constraint would be built as follows:

```
ClLinearInequality cn =
        new ClLinearInequality(CL.Plus(CL.Times(x,3),5), CL.LEQ, y);
```

Although the Java implementation makes it more difficult to express programmer-written constraints, this inconvenience is relatively unimportant when the solver is used in conjunction with graphical user interfaces for specifying the constraints.

A.2.5 *Symbolic Weights and Strengths.* The constraint hierarchy theory allows an arbitrary (although finite) number of strengths of constraint. In practice, however, programmers use a small number of strengths in a stylized way. The current implementation therefore includes a small number of pre-defined strengths, and the maximum number of strengths is defined as a constant. (This constant can be changed as discussed below, but we would not expect to do so frequently.)

The strengths provided in the current release are:

required Required constraints must be satisfied and should be used only for relationships that make no sense unless they are exactly met. The most common

use of a required constraint is to give a shorthand name to an expression such as: win.right = win.left + win.width

strong This strength is conventionally used for edit constraints.

medium This strength can be used for strong stays constraints; for example, we might put medium strength stay constraints on the width and height of an object and weak stay constraints on its position to represent our preference that the object move instead of change size when either is possible to maintain the stronger constraints.

weak This strength is used for stay constraints.

Each strength category is represented as an instance of ClStrength.

A related class is ClSymbolicWeight. As mentioned in Section 2.3, the objective function is formed as the weighted sum of the positive and negative errors for the non-required constraints. The weights should be such that the stronger constraints totally dominate the weaker ones. In general to pick a real number for the weight we need to know how big the values of the variables can be. To avoid this problem altogether, in the Smalltalk and C++ implementations we use symbolic weights and a lexicographic ordering for the weights rather than real numbers, which ensures that strong constraints are always satisfied in preference to weak ones.

Instances of ClSymbolicWeight are used to represent these symbolic weights. These instances have an array of floating point numbers, whose length is the number of non-required strengths (so 3 at the moment). Each element of the array represents the value at that strength, so $[1.0, 0.0, 10.0]$ represents a weight of 1.0 strong, 0.0 medium, and 10.0 weak. (In Smalltalk ClSymbolicWeight is a variable length subclass; we could have had an instance variable with an array of length 3 instead.) Symbolic weights understand various arithmetic messages (or operator overloading in C++) as follows:

+ w
w is also a symbolic weight. Return the result of adding w to self (or this in C++).

– w
w is also a symbolic weight. Return the result of subtracting w from self.

* n
n is a number. Return the result of multiplying self by n.

/ n
n is a number. Return the result of dividing self by n.

<= n, >= n, < n, > n, = n
w is a symbolic weight. Return true if self is related to n as the operator normally queries.

negative
Return true if this symbolic weight is negative (i.e., it does not consist of all zeros and the first non-zero number is negative).

Instances of ClStrength represent a strength in the constraint hierarchy. The instance variables are name (for printing purposes) and symbolicWeight, which is

the unit symbolic weight for this strength. Thus, with the 3 strengths as above, strong is $[1.0, 0.0, 0.0]$, medium is $[0.0, 1.0, 0.0]$, and weak is $[0.0, 0.0, 1.0]$.

The above arithmetic messages let the Smalltalk implementation of the solver use symbolic weights just like numbers in expressions. This interface is important because the objective row in the tableau has coefficients which are ClSymbolicWeights but are subject to the same manipulations as the other tableau rows whose expressions have coefficients that are just real numbers.

In both C++ and Java, an additional message asDouble() is understood by ClSymbolicWeights. This converts the representation to a real number that approximates the total ordering suggested by the more general vector of real numbers. It is these real numbers that are used as the coefficients in the objective row of the tableau instead of ClSymbolicWeights (which the coefficients conceptually are). This kludge avoids the complexities that such genericity introduces to the static type systems of C++ and Java.

Also, since Java lacks operator overloading, the above operations are invoked using suggestive alphabetic method names such as add, subtract, times, and lessThan.

A.3 ClSimplexSolver Implementation

Here are the instance variables of ClSimplexSolver (some fields are inherited from ClTableau, the base class of ClSimplexSolver which provides the basic sparse-matrix interface—see Section A.3.1).

rows
: A dictionary with keys ClAbstractVariable and values ClLinearExpression. This holds the tableau. Note that the keys can be either restricted or unrestricted variables, i.e., both C_U and C_S are actually merged into one tableau. This simplified the code considerably, since most operations are applied to both restricted and unrestricted rows.

columns
: A dictionary with keys ClAbstractVariable and values Set of ClAbstractVariable. These are the column cross-indices. Each parametric variable p should be a key in this dictionary. The corresponding set should include exactly those basic variables whose linear expression includes p (p will of course have a non-zero coefficient). The keys can be either unrestricted or restricted variables.

objective
: An instance of ClObjectiveVariable (named z) that is the key for the objective row in the tableau.

infeasibleRows
: A set of basic variables that have infeasible rows. (This field is used when re-optimizing with the dual simplex method.)

prevEditConstants
: An array of constants (floats) for the edit constraints on the previous iteration. The elements in this array must be in the same order as editPlusErrorVars and editMinusErrorVars, and the argument to the public resolve: message.

stayPlusErrorVars, stayMinusErrorVars
: An array of plus/minus error variables (instances of ClSlackVariable) for the

stay constraints. The corresponding negative/positive error variable must have the same index in stayMinusErrorVars/stayPlusErrorVars.

editPlusErrorVars, editMinusErrorVars
: An array of plus/minus error variables (instances of ClSlackVariable) for the edit constraints. The corresponding negative/positive error variable must have the same index in editMinusErrorVars/editPlusErrorVars.

markerVars
: A dictionary whose keys are constraints and whose values are instances of a subclass of ClAbstractVariable. This dictionary is used to find the marker variable for a constraint when deleting that constraint. A secondary use is that iterating through the keys will give all of the original constraints (useful for the reset method).

errorVars
: A dictionary whose keys are constraints and whose values are arrays of ClSlackVariable. This dictionary gives the error variable (or variables) for a given non-required constraint. We need this if the constraint is deleted because the corresponding error variables must be deleted from the objective function.

A.3.1 ClTableau *(Sparse Matrix) Operations.* The basic requirements for the tableau representation are that one should be able to perform the following operations efficiently:

—determine whether a variable is basic or parametric

—find the corresponding expression for a basic variable

—iterate through all the parametric variables with non-zero coefficients in a given row

—find all the rows that contain a given parametric variable with a non-zero coefficient

—add/remove a row

—remove a parametric variable

—substitute out a variable (i.e., replace all occurrences of a variable with an expression, updating the tableau as appropriate).

The representation of the tableau as a dictionary of rows, with column cross-indices, supports these operations. Keeping the cross indices up-to-date and consistent with the row dictionary is error-prone. Thus, the solver actually accesses the rows and columns only via the below interface of ClTableau.

addRow(ClAbstractVariable var, ClLinearExpression expr)
: Add the constraint var=expr to the tableau. var will become a basic variable. Update the column cross indices.

noteAddedVariable(ClAbstractVariable var, ClAbstractVariable subject)
: Variable var has been added to the linear expression for subject. Update the column cross indices.

noteRemovedVariable(ClAbstractVariable var, ClAbstractVariable subject)
: Variable var has been removed from the linear expression for subject. Update the column cross indices.

removeColumn(ClAbstractVariable var)
: Remove the parametric variable var from the tableau. This operation involves removing the column cross index for var and removing var from every expression in rows in which it occurs.

removeRow(ClAbstractVariable var)
: Remove the basic variable var from the tableau. Because var is basic, there should be a row var=expr. Remove this row, and also update the column cross indices.

substituteOut(ClAbstractVariable var, ClLinearExpression expr)
: Replace all occurrences of var with expr and update the column cross indices.

A.4 Omissions

The solver should implement Bland's anti-cycling rule [Marriott and Stuckey 1998], but it does not at the moment. Adding this should be straightforward.

B. COMPARATOR DETAILS

Our implementation of Cassowary favors solutions that satisfies some of the constraints completely, rather than ones that, for example, partially satisfy each of two conflicting equalities. These are still legitimate locally-error-better and weighted-sum-better solutions. Cassowary's behavior is analogous to that of the simplex algorithm, which always finds solutions at a vertex of the polytope even if all the solutions on an edge or face are equally good. (Of course, Cassowary behaves this way because the simplex algorithm does.)

Such solutions are also produced by greedy constraint satisfaction algorithms, including local propagation algorithms such as DeltaBlue [Sannella et al. 1993] and Indigo [Borning et al. 1996]: these algorithms try to satisfy constraints one at a time, and in effect the constraints considered first are given a stronger strength than those considered later.

However, there is an issue regarding comparators and Cassowary that has not yet been resolved in an entirely clean way. One of the public methods for Cassowary is addPointStays: points, as discussed in Section A.2. This method addresses the desire to satisfy the stays on both the x and y components of a given point rather than on the x component of one point and the y component of another.

As an example of why this is useful, consider a line with endpoints p1 and p2 and a midpoint m. There are constraints (p1.x+p2.x)/2 = m.x and (p1.y+p2.y)/2 = m.y. Suppose we are editing m. It would look strange to satisfy the stay constraints on p1.x and p2.y, rather than both stays on p1 or both stays on p2. (This claim has been verified empirically—in earlier implementations of Cassowary this happened, and indeed it looked strange.)

The current implementation of addPointStays: points uses different weights for the stay constraints for successive elements of points—a kludge that seems to work well in practice.

It was difficult to devise an example where it would give a bad answer—here is a contrived one. Suppose we have a line with endpoints p1 and p2 and a midpoint m. Suppose also we have constraints p2.x = 2*p3.x and p2.y = 2*p3.y. (This example is a bit strange since here we are using p3 as a distance from the origin rather than

as a location—otherwise multiplying it by 2 is problematic.) If we give these points to addPointStays: in the order p1, p2, and p3, then the stays on p1 will have weight 1, those on p2 will have weight 0.5, and those on p3 will have weight 0.25. Then, a one legitimate WSB solution would satisfy the stays on p1.x and p1.y, but another legitimate WSB solution would satisfy the stays on p1.x, p2.y, and p3.y.

Here is a cleaner way to handle this situation. We first introduce a new comparator with the dubious name of *tilted-locally-error-better*. The set of TLEB solutions can be defined by taking a given hierarchy, forming all possible hierarchies by breaking strength ties in all possible ways to form a totally ordered set of constraints, and taking the union of the sets of solutions to each of these totally ordered hierarchies.

For example, consider the two constraints weak $x = 0$ and weak $x = 10$. The set of LEB solutions is the infinite set of mappings from x to each number in [0...10]. Assuming equal weights on the constraints, the (single) least-squares solution is $\{x \mapsto 5\}$. The TLEB solutions are defined by producing all the totally ordered hierarchies and taking the union of their solutions. In this case the two possible total orderings are:

weak $x = 0$, slightly_weaker $x = 10$
slightly_weaker $x = 0$, weak $x = 10$

These have solutions $\{x \mapsto 0\}$ and $\{x \mapsto 10\}$ respectively, so the set of TLEB solutions to the original hierarchy is $\{\{x \mapsto 0\}, \{x \mapsto 10\}\}$. As an aside, we hypothesize that the only psychologically plausible solutions to the example are $\{x \mapsto 0\}$, $\{x \mapsto 5\}$, and $\{x \mapsto 10\}$, but not, for example, $\{x \mapsto 3.8\}$.

Next, we introduce a notion of a *compound constraint*, a conjunction of primitive constraints, in this case linear equalities or inequalities. For compound constraints, when we break the strength ties in defining the set of tilted-locally-error-better solutions, we insist on mapping each linear equality or inequality in a compound constraint to an adjacent strength. (We have been a bit imprecise in the use of the term "constraint" in this paper, sometimes using it to denote a primitive constraint and sometimes to denote a conjunction of primitive constraints. For the present definition, however, we need to distinguish compound constraints that have been specifically identified as such by the user from conjunctions of primitive constraints more generally, such as the constraints C_S and C_U discussed in Section 2.1.)

Now, to define addPointStays: in a more clean way, we could make each point stay a compound constraint. To illustrate why this works, consider the midpoint example again. We have two endpoints p1 and p2, and a midpoint m. There are constraints (p1.x+p2.x)/2 = m.x and (p1.y+p2.y)/2 = m.y, and we are editing m. Then the stays on p1 and p2 will each be compound constraints:

weak (*stay* $p1.x$ & *stay* $p1.y$)
weak (*stay* $p2.x$ & *stay* $p2.y$)

In defining the set of tilted-locally-error-better solutions, the total orderings of these constraints that we will consider have the stays on $p1.x$ and $p1.y$ both stronger than those on $p2.x$ and $p2.y$, or both weaker. Thus, we produce the desired result.

It is not sufficient just to define a notion of "compound constraint" without adding the notion of tilting—otherwise if we were using locally-error-better, we would just sum the errors of the primitive constraints which would allow us to

trade off the errors arbitrarily; we could still satisfy the stay on the x component of one point and the y component of another.

Regarding other comparators, tilting is incompatible with the weighted-sum comparator. Using tilting to break a tie between two constraints with the same strength and weight could lead to incorrect results because one of the constraints would dominate a third constraint with the same strength and larger weight. In any case, the weights already can be used to break ties among constraints with the same strength.

REFERENCES

Badros, G. J. and Borning, A. 1999. Cassowary constraint solving toolkit. Web page. http://www.cs.washington.edu/research/constraints/cassowary.

Badros, G. J., Borning, A., Marriott, K., and Stuckey, P. 1999. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology* (November 1999).

Badros, G. J., Nichols, J., and Borning, A. 2000. Scwm—the Scheme Constraints Window Manager. In *Proceedings of the AAAI Spring Symposium on Smart Graphics* (March 2000). To appear.

Badros, G. J. and Stachowiak, M. 1999. Scwm—The Scheme Constraints Window Manager. Web page. http://scwm.mit.edu/scwm/.

Baraff, D. 1994. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94 Conference Proceedings* (1994), pp. 23–32. ACM.

Borning, A., Anderson, R., and Freeman-Benson, B. 1996. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology* (Seattle, Nov. 1996), pp. 129–136.

Borning, A. and Freeman-Benson, B. 1995. The OTI constraint solver: A constraint library for constructing interactive graphical user interfaces. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming* (Cassis, France, Sept. 1995), pp. 624–628.

Borning, A., Freeman-Benson, B., and Wilson, M. 1992. Constraint hierarchies. *Lisp and Symbolic Computation 5*, 3 (Sept.), 223–270.

Borning, A., Lin, R., and Marriott, K. 1997. Constraints for the web. In *Proceedings of ACM MULTIMEDIA'97* (Nov. 1997).

Borning, A., Marriott, K., Stuckey, P., and Xiao, Y. 1997. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology* (Oct. 1997).

Helm, R., Huynh, T., Lassez, C., and Marriott, K. 1992. A linear constraint technology for interactive graphic systems. In *Graphics Interface '92* (1992), pp. 301–309.

Helm, R., Huynh, T., Marriott, K., and Vlissides, J. 1992. An object-oriented architecture for constraint-based graphical editing. In *Proceedings of the Third Eurographics Workshop on Object-oriented Graphics* (Champery, Switzerland, Oct. 1992).

Hosobe, H., Matsuoka, S., and Yonezawa, A. 1996. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming* (Boston, Aug. 1996).

Hudson, S. and Smith, I. 1996. SubArctic UI toolkit user's manual. Technical report, College of Computing, Georgia Institute of Technology.

Huynh, T. and Marriott, K. 1995. Incremental constraint deletion in systems of linear constraints. *Information Processing Letters 55*, 111–115.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems 14*, 3 (July), 339–395.

Marriott, K. and Stuckey, P. 1998. *Programming with Constraints: An Introduction*. MIT Press.

Myers, B. A. 1996. The Amulet user interface development environment. In *CHI'96 Conference Companion: Human Factors in Computing Systems* (Vancouver, B.C., April 1996).

ACM SIGCHI.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. 1989. *Numerical Recipes: The Art of Scientific Computing* (second ed.). Cambridge University Press.

Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. 1993. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience 23*, 5 (May), 529–566.

Sutherland, I. 1963. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference* (1963), pp. 329–346. IFIPS.

Vander Zanden, B. 1996. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Transactions on Programming Languages and Systems 18*, 1 (Jan.), 30–72.

Wolfman, S. and Weld, D. 1998. The LPSAT engine and its application to resource planning. In *Proceedings of IJCAI'99* (1998).