Solving Linear Arithmetic Constraints for User Interface Applications: Algorithm Details

Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao

Technical Report 97-06-01
Department of Computer Science & Engineering
University of Washington
September 1997

Abstract. 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 1/3 of a window, or preferring that an object be contained within a rectangle if possible. Current constraint solvers designed for UI applications cannot efficiently handle simultaneous linear equations and inequalities. This is a major limitation. We describe incremental algorithms based on the dual simplex and active set methods that can solve such systems of constraints efficiently. Both algorithms have been implemented and tested.

This informal technical report is adapted from the paper "Solving Linear Arithmetic Constraints for User Interface Applications," which will appear in the Proceedings of UIST'97 (The ACM User Interface and Software Technology Symposium). It contains additional details, in particular of the Cassowary and QOCA algorithms.

Authors' addresses:

Alan Borning
Dept. of Computer Science & Engr.
University of Washington
PO Box 352350
Seattle, Washington 98195, USA
borning@cs.washington.edu

Kim Marriott
Dept. of Computer Science
Monash University
Wellington Road
Clayton, Victoria 3168, Australia
marriott@cs.monash.edu.au

Peter Stuckey
Dept. of Computer Science
University of Melbourne
Parkville, Victoria 3052, Australia
pjs@cs.mu.oz.au

Yi Xiao
Dept. of Mathematics & Statistics
University of Melbourne
Parkville, Victoria 3052, Australia
yxiao@maths.mu.oz.au

1 Introduction

Linear equality and inequality constraints arise naturally in specifying many aspects of user interfaces, in particular 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. One use is 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 is to process potentially invalid user inputs in a graceful way. 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 given an error. A third use is to balance conflicting desires, for example in laying out a graph.

Efficient techniques are available for solving such 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 in fact often cyclic. This sometimes arose when the programmer added redundant constraints — the cycles *could* have been avoided by careful analysis. However, this 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.

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* [4] 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. 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 choice 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, for example the reals, can have an associated error function. The error in satisfying a constraint *cn* is 0 iff the constraint is satisfied, and becomes larger the less nearly satisfied is the constraint.)

As described in [2], for inequality constraints it is important to use a metric rather than a predicate comparator. 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; QOCA finds a least-squares-better solution. The least-squares-better comparator strongly penalizes outlying values when trading off constraints of the same strength. It is particularly suited to tasks such as

laying out a tree, a graph, or a collection of windows, where there are inherently conflicting preferences (for example, that all the nodes in the depiction of a graph have some minimum spacing and that edge lengths be minimized). Locally-error-better, on the other hand, is a more permissive comparator, in that it admits more solutions to the constraints. (In fact any least-squares-better or weighted-sum-better solution is also a locally-error-better solution [4].) It is thus easier to implement algorithms to find a locally-error-better solution, and in particular to design hybrid algorithms that include subsolvers for simultaneous equations and inequalities and also subsolvers for nonnumeric constraints [3]. Since each of these different comparators is preferable in certain situations we give algorithms for each.

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 $1 \leq i \leq n$. 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, we wish to find values for the x_i that minimizes (or maximizes) the value of the *objective function*

$c + d_1x_1 + \ldots + d_nx_n$.

This problem has been heavily studied for the past 50 years. The most commonly used algorithm for solving it is the simplex algorithm, developed by Dantzig in the 1940s, and there are now numerous variations of it. Unfortunately, however, existing implementations of the simplex are not really suitable for UI applications.

The principal issue is incrementality. For interactive graphical applications, we need to solve similar problems repeatedly, rather than solving a single problem once, and to achieve interactive response times, very fast incremental algorithms are needed. There are two cases. 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 this case we must re-satisfy the same collection of constraints, differing only in the mouse location, each time the screen is refreshed. Second, when editing an object we may add or remove constraints and other parts, and we would like to make these operations fast, by reusing as much of the previous solution as possible. The performance requirements are considerably more stringent for the first case than for the second.

Another issue is defining a suitable objective function. 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. Fortunately techniques have been developed in the operations research community for handling these cases, which we adopt here. For the first two comparators, the objective functions are "almost linear," while the third comparator gives rise to a quadratic optimization problem.

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

1.3 Overview

We present algorithms for incrementally solving linear equality and inequality constraints for the three different comparators described above. In Section 2.1 we give algorithms for incrementally adding and deleting required constraints with restricted and unrestricted variables from a system of constraints kept in *augmented simplex form*, a type of solved form. In Section 3.1 we give an algorithm, Cassowary, based on the dual simplex, for incrementally solving hierarchies of constraints using the locally-error-better or weighted-sum-better comparators when a constraint is added or an object is moved, while in Section 5 we give an algorithm, QOCA, based on the active set method, for incrementally solving hierarchies of constraints using the least-squares-better comparator.

Both of our algorithms have been implemented, Cassowary initially in Smalltalk and QOCA in C++. They perform surprisingly well, and a summary of our results is given in Section 6. The QOCA implementation is considerably more sophisticated and has much better performance than the current version of Cassowary. However, QOCA is inherently a more complex algorithm, and re-implementing it with a comparable level of performance would be a daunting task. In contrast, Cassowary is straightforward, and a reimplementation based on this paper is more reasonable, given a knowledge of the simplex algorithm. In fact we have recently re-implemented Cassowary in Java.

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 [19]. Most of the current systems use one-way constraints (e.g. [12, 16]), or local propagation algorithms for acyclic collections of multi-way constraints (e.g. [18, 20]). Indigo [2] handles acyclic collections of inequality constraints, but not cycles (simultaneous equations and inequalities). UI systems that handle simultaneous linear equations include DETAIL [11] and Ultraviolet [3]. A number of researchers (including the first 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. An earlier version of QOCA is described in references [9] and [10]. These earlier descriptions, however, do not give any details of the algorithm, although the incremental deletion algorithm is described in [13]. The current implementation is much improved, in particular through the use of the active set method described in Section 5.2.

Baraff [1] describes a quadratic optimization algorithm for solving linear constraints that arise in modelling physical systems. Finally, much of the work on constraint solvers has been in the logic programming and constraint logic programming communities. Current constraint logic programming languages such as CLP($\mathcal{R}$) [14] include efficient solvers for linear equalities and inequalities. (See [15] 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 notion as such of state and change. As a result, these systems are not so well suited for building interactive graphical applications.

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*, [17, page 424]

We now describe an incremental version of the simplex algorithm, adapted to the task at hand. The material presented in this section is common to both Cassowary and QOCA. The two algorithms use different optimization techniques, however, which are described in Sections 3 and 5 respectively. In the description we use a running example, illustrated by the diagram in Figure 1.

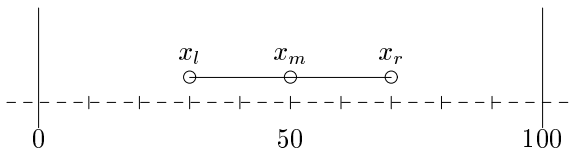

Figure 1: Simple constrained picture

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 0 to 100. (To keep the presentation manageable, we deal only with the x coordinates. Adding analogous constraints on the y coordinates would be simple but would double the number of the 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.

We can represent this using the constraints

$$\begin{array}{rcl} 2x_m & = & x_l + x_r \\ x_l + 10 & \leq & x_r \\ x_r & \leq & 100 \\ 0 & \leq & x_l \end{array}$$

Now suppose we wish to minimize the distance between x_m and x_l or in other words, minimize $x_m - x_l$.

2.1 Augmented Simplex Form

An optimization problem is in *augmented simplex form* if constraint C has the form $C_U \wedge C_S \wedge C_I$ where C_U and C_S are conjunctions of linear arithmetic equations and 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 does not itself handle variables that may take negative values (so-called *unrestricted variables*), and imposes a 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 [15], and we have adopted it here. Essentially it uses *two* tableaux rather than one. All of the unrestricted variables will be placed in C_U, the unrestricted variable tableau. C_S, the simplex tableau, contains only variables constrained to be non-negative. The simplex algorithm is used to determine an optimal solution for the equations in the simplex tableau, ignoring the unrestricted variable tableau for purposes of optimization. The equations in the unrestricted variable tableau are then used to determine values for its variables.

Implementation Note. In the paper we describe C_U and C_S as two separate tableaux. In the implementation, however, it turns out to be simpler to have just one tableau, since most operations are applied to both C_U and C_S. Unrestricted and restricted variables are instances of different classes, and in the code we differentiate when necessary by sending the is_restricted message to the variable for each row. See Section 4.

It is not difficult to 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 new non-negative *slack* variable.

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

minimize $x_m - x_l$ subject to

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

We now separate the equalities into C_U and C_S. Initially all equations are in C_S. We separate out 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 the third equation can be used to substitute $100 - s_2$ for x_r obtaining

minimize $x_m - x_l$

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

Next, and the first equation can be used to substitute $50 + \frac{1}{2}x_l - \frac{1}{2}s_2$ for x_m, giving

minimize $50 - \frac{1}{2}x_l - \frac{1}{2}s_2$ subject to

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

The tableau shows C_U above the horizontal line, and C_S and C_I below the horizontal line. From now on C_I will be omitted — any variable occurring below the horizontal line is implicitly constrained to be non-negative. The simplex method works by taking a 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.

A Detail. The example includes the constraint $x_l \geq 0$. To simplify the example, we just make x_l be a restricted variable to capture this constraint. In the Cassowary implementation, however, all variables that may be accessed from outside the solver as well as within it are unrestricted. Only error or slack variables are represented as restricted variables, and these variables occur only within the solver. See Section 4. The primary benefit of this 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.)

Another Detail. The operations are shown 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 at least for Cassowary it would make the incremental updates of the constants in edit constraints slower; and since this is a much more frequent operation, in the Cassowary implementation we do actually modify both C_U and C_S.

A 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 $50 - \frac{1}{2}x_l + \frac{1}{2}s_2$ subject to

$$\begin{array}{rcrrr} x_m & = & 50 & +\frac{1}{2}x_l & -\frac{1}{2}s_2 \\ x_r & = & 100 & & -s_2 \\ \hline s_1 & = & 90 & -x_l & -s_2 \end{array}$$

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

$$\{x_m \mapsto 50, x_r \mapsto 100, s_1 \mapsto 90, x_l \mapsto 0, s_2 \mapsto 0\}.$$

The value of the objective function with this solution is 50.

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. When no such adjacent basic feasible solved form can be found, the optimum has been found. 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 x_l from 0 will decrease the value of the objective function. We must be careful as we cannot increase the value of x_l indefinitely as this may cause the value of some other basic non-negative variable to become negative. We must examine the equations in C_S. The equation $s_1 = 90 - x_l - s_2$ allows x_l to take at most a value of 90, as if x_l becomes larger than this, then s_1 would become negative. The equations above the horizontal line do not restrict x_l, since whatever value x_l takes the unrestricted variables x_m and x_r can take a value to satisfy the equation. In general, we choose the most restrictive equation in C_S, and use it to eliminate x_l. In the case of ties we arbitrarily break the tie. In this example the most restrictive equation is $s_1 = 90 - x_l - s_2$.

Writing x_l as the subject we obtain $x_l = 90 - s_1 - s_2$. We replace x_l everywhere by $90 - s_1 - s_2$ and obtain

minimize $5 + \frac{1}{2}s_1 + s_2$ subject to

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

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

We continue this process. Increasing the value of s_1 will increase the value of the objective. Note that decreasing s_1 will also decrease the objective function 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.

(If we were to have an unrestricted variable in the objective function, the optimization would be unbounded. This is not an issue for Cassowary or QOCA, since the objective function in those cases always only contains restricted variables, i.e. variables implicitly constrained to be non-negative.)

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.

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

$$\begin{array}{rcl} \bigwedge_{i=1}^{n} x_i & = & c_i + \sum_{j=1}^{m} a_{ij} y_j \wedge \\ \bigwedge_{i=1}^{n} x_i \geq 0 \wedge \bigwedge_{j=1}^{m} y_j \geq 0. & & \end{array}$$

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. This is because we could choose y_J to take an arbitrarily large value, and so 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 illustrated in Figure 2, and takes as inputs the simplex tableau C_S and the objective function f.

2.3 Incrementality: Adding a Constraint

We now describe how to add the equation for a new constraint incrementally. This technique is also used in our implementations 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.

```
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

As an example, suppose we wish to ensure that the midpoint sits in the centre of the screen. This is represented by the constraint $x_m = 50$. If we substitute for each of the basic variables (only x_m) in this constraint we obtain the equation $45 - \frac{1}{2}s_1 - s_2 = 0$. In order to add this constraint straightforwardly to the tableau we create a new non-negative variable a called an *artificial variable*. (This is simply an incremental version of the operation used in Phase I of the two-phase simplex algorithm.) We let $a = 45 - \frac{1}{2}s_1 - s_2$ 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

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

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 and 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.

Implementation Note. In some cases we can add an equation to the tableau without using an artificial variable, and for efficiency should do so when it is easy to detect that this can be done. See Section 4.3.2.

2.4 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 influence in the tableaux by looking for occurrences of that marker variable. For inequality constraints, the slack variable s added to make it an equality serves as the marker, since s will originally occur only in that equation. The equation representing a nonrequired equality constraint will have an error variable that can serve as a marker — see Section 2.5. 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 add a dummy variable s_3, resulting in $2x_m = x_l + x_r + s_3$. The simplex optimization routine checks for these dummy variables in choosing an entry variable, and does not allow one to be selected. (We didn't include this variable in the tableaux presented earlier to keep things simpler.)

(Note: these dummy variables must be restricted, not unrestricted, since we might need to have some of them in the equations for restricted basic variables.)

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 above s_1 is already basic, and so removing it simply means dropping the row in which it is basic, obtaining

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

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), 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 and using that to pivot s_1 into the basis, and then remove the row.

There is such a restrictive equation in this example. However, if no equation restricts the size of the marker variable, that is, its coefficients are all non-negative, then either the marker variable has a positive coefficient in all equations, or it only occurs in equations for unrestricted variables. If it does occur in an equation for a restricted variable, pick the equation that gives the smallest ratio. (The row with the marker variable will become infeasible, 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 it only occurs in equations for unrestricted variables, we can choose any equation in which it occurs.

In the case above, the row $x_l = 90 - s_1 - s_2$ 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

$$\begin{array}{rcrrr} x_m & = & 50 & +\frac{1}{2}x_l & -\frac{1}{2}s_2 \\ x_r & = & 100 & & -s_2 \\ \hline \end{array}$$

In the preceding example the marker variable had a negative coefficient. Here is an example in which it just has positive coefficients. (This is an example just for the tech report.) 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:

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

where s_1, s_2, and s_3 are the marker variables for $x \geq 10$, $x \geq 20$, and $x \geq 30$ respectively.

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:

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

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

$$\begin{array}{rcrr} x & = & 20 & +s_2 \\ \hline s_l & = & 10 & +s_2 \end{array}$$

which is of course feasible.

As another fine point, note that there is no problem with redundant constraints. 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.

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

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:

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

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

A consequence of this is that if there are two redundant constraints, both of them must be removed to eliminate their effect. (This seems to be a more desirable behaviour 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. (If there weren't dummy marker variables, we *would* do this for redundant required equality constraints.)

2.5 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 x_m *edit*, x_l *stay*, and x_r *stay*. 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 $x_r = 60$. There are two possible translations of these non-required constraints to an objective function, depending on the comparator used.

For locally-error-better or weighted-sum-better, we can simply add the errors of the each constraint to form an objective function. Consider the constraint $x_m = 50$. We define the error as $|x_m - 50|$. We need to combine the errors for each non-required constraint with a weight so we obtain the objective function

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

where s and w are weights so that the strong constraint is always strictly more important than solving any combination of weak constraints, so that we find a locally-error-better or weighted-sum-better solution. For the least-squares-better comparator the objective function is

$$s(x_m - 50)^2 + w(x_l - 30)^2 + w(x_r - 60)^2.$$

In the presentation, we will use $s = 1000$ and $w = 1$.

Cassowary actually uses symbolic weights and a lexicographic ordering, which ensures that strong constraints are always satisfied in preference to weak ones (see Section 4). However, QOCA does not employ symbolic weights.

Unfortunately neither of these objective functions is linear and hence the simplex method is not applicable directly. We now show how we can solve the problem using optimization algorithms based on the two alternate objective functions: *quasi-linear optimization* and *quadratic optimization*.

3 Cassowary: Quasi-linear Optimization

Cassowary finds either locally-error-better or weighted-sum-better solutions. Since every weighted-sum-better solution is also a locally-error-better solution [4], the weighted-sum part of the optimization comes automatically from the manner in which the objective function is constructed.

For Cassowary 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. Since 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 for 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:

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

The objective function to satisfy the non-required constraints can now be restated as

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

An optimal solution of this problem can be found using the simplex algorithm, and results in a tableau

minimize $10 + 1002\delta_{x_m}^+ + 998\delta_{x_m}^- + 2\delta_{x_l}^- + 2\delta_{x_r}^-$ subject to

$$\begin{array}{rcl|rrrrr} x_m & = & 50 & +\delta_{x_m}^+ & -\delta_{x_m}^- & & & \\ x_r & = & 70 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -\delta_{x_l}^+ & +\delta_{x_l}^- & \\ \hline x_l & = & 30 & & & +\delta_{x_l}^+ & -\delta_{x_l}^- & \\ s_1 & = & 30 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -2\delta_{x_l}^+ & +2\delta_{x_l}^- & \\ s_2 & = & 30 & -2\delta_{x_m}^+ & +2\delta_{x_m}^- & +\delta_{x_l}^+ & -\delta_{x_l}^- & \\ \delta_{x_r}^+ & = & 10 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -\delta_{x_l}^+ & +\delta_{x_l}^- & +\delta_{x_r}^- \end{array}$$

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.

3.1 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 reoptimizing 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 last equation (the equation for $\delta_{x_r}^+$) to 0.

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

minimize $20 + 1002\delta_{x_m}^+ + 998\delta_{x_m}^- + 2\delta_{x_l}^- + 2\delta_{x_r}^-$ subject to

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

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.

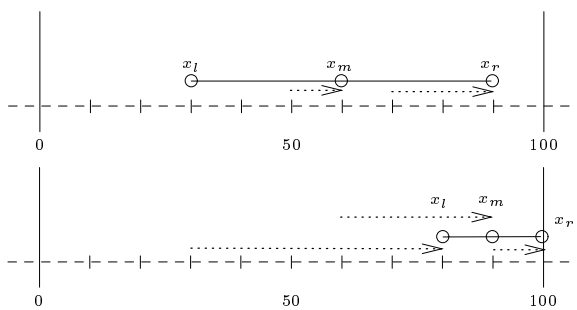

Figure 3: Resolving the constraints

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

minimize $60 + 1002\delta_{x_m}^+ + 998\delta_{x_m}^- + 2\delta_{x_l}^- + 2\delta_{x_r}^-$ subject to

$$\begin{array}{rcrrrrrr}
x_m & = & 90 & +\delta_{x_m}^+ & -\delta_{x_m}^- & & & \\
x_r & = & 150 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -\delta_{x_l}^+ & +\delta_{x_l}^- & \\
\hline
x_l & = & 30 & & & +\delta_{x_l}^+ & -\delta_{x_l}^- & \\
s_1 & = & 110 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -2\delta_{x_l}^+ & +2\delta_{x_l}^- & \\
s_2 & = & -50 & -2\delta_{x_m}^+ & +2\delta_{x_m}^- & +\delta_{x_l}^+ & -\delta_{x_l}^- & \\
\delta_{x_r}^+ & = & 60 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -\delta_{x_l}^+ & +\delta_{x_l}^- & +\delta_{x_r}^-
\end{array}$$

The tableau is no longer in basic feasible solved form, since the constant of the row for s_2 is negative, even though s_2 is supposed to be non-negative.

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. And since no coefficient has changed, in particular in the optimization function, 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 we did when adding a constraint), optimizing the sum of the artificial variables to find an initial feasible solution, and then reoptimizing 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

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

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

where some c_i may be negative for rows with non-negative basic variables (infeasibility) and each d_j is non-negative (optimality).

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 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_2, 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 the entry variable. Replacing $\delta_{x_l}^{+}$ everywhere by $50 + s_2 + 2\delta_{x_m}^{+} - 2\delta_{x_m}^{-} + \delta_{x_l}^{+}$ we obtain the tableau

minimize $30060 + 1002\delta_{x_m}^{+} + 998\delta_{x_m}^{-} + 2\delta_{x_l}^{-} + 2\delta_{x_r}^{-}$ subject to

$$\begin{array}{rcrrrrrr}
x_m & = & 90 & & +\delta_{x_m}^{+} & -\delta_{x_m}^{-} & & \\
x_r & = & 100 & -s_2 & & & & \\
\hline
x_l & = & 80 & +s_2 & +2\delta_{x_m}^{+} & -2\delta_{x_m}^{-} & & \\
s_1 & = & 110 & -2s_2 & +2\delta_{x_m}^{+} & -2\delta_{x_m}^{-} & & \\
\delta_{x_l}^{+} & = & 50 & +s_2 & +2\delta_{x_m}^{+} & -2\delta_{x_m}^{-} & +\delta_{x_l}^{-} & \\
\delta_{x_r}^{+} & = & 40 & -s_2 & & & & +\delta_{x_r}^{-}
\end{array}$$

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 constraints, then updating the constants to

```
re_opt(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

reflect the updated edit constraints, and finally reoptimizing if needed. In an interactive graphical application, when using the dual optimization method typically a pivot is only required when one part first hits a barrier, 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; 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 illustrates why the dual optimization is well suited to this problem and leads to efficient resolving of the hierarchical constraints.

4 Cassowary Details

This section is only in the tech report (not the UIST paper) and includes details on the Cassowary implementation. The current implementation is in Smalltalk. However, it should be straightforward to translate into another object-oriented language. There is also a subsection on some fine points regarding the comparator.

4.1 Solver Protocol

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

addConstraint: cn
: Incrementally add the linear constraint cn to the tableau.

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

resolve: newEditConstants
: newEditConstants is an array of floating point numbers, which are new values for the constants in the edit constraints. The size of this array must be the same as the number of edit constraints currently in the tableau. The order of the array elements should be the same as the order in which the edit constraints were added.

addPointStays: points
: This is kind 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. points is an array of points, whose x and y components are constrainable variables. Add 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 Subsection 4.5 for more on this issue.

reset
: Re-initialize the solver from the original constraints, thus getting rid of any accumulated numerical problems. (It's not clear how often such problems arise, but here is the method anyway.)

4.1.1 Possible Revisions to Solver Protocol

One thing that might be worth changing is the way that edit and stay constraints are added. Currently edit and stay constraints are added using the addConstraint: message, just as with any other kind of constraint. Edit and stay constraints can be added or deleted at any time.

The order in which the edit constraints are added is critical, however, since it corresponds to the order in which the floats are given for the resolve: message. In addition, there is nothing that requires the programmer to remove the old edit constraints before adding new ones.

Also, if variables have a value (before doing any constraint solving), one should add stay constraints on these variables before adding other constraints, since otherwise the variable's value is likely to be changed inappropriately to satisfy the other constraints.

Given this, an alternative would be to have a separate editConstraints: message, which added all the edit constraints at once (and at the same time removed any old ones). Stay constraints would not be represented explicitly, but instead would be implicit for each variable. Thus the stay constraints would in effect be added before any other constraints. See Subsection 4.2.4.

4.2 Principal Classes

Here is a listing of the principal classes. (In the current implementation all the classes start with "Cl".) All of the classes are of course direct or indirect sublclasses of Object.

- Object
  - ClAbstractVariable
    - ClDummyVariable
    - ClObjectiveVariable
    - ClSlackVariable
    - ClVariable
  - ClConstraint
    - ClEditOrStayConstraint
      - ClEditConstraint
      - ClStayConstraint
    - ClLinearConstraint
      - ClLinearEqualityConstraint
      - ClLinearInequalityConstraint
  - ClLinearExpression
  - ClSimplexSolver
  - ClStrength
  - ClSymbolicWeight

Following is a description of the classes. Some of these classes make use of a class Dictionary, which is part of the Smalltalk system. Dictionaries have keys and values. There is a hash table that lets one efficiently find the value for a given key, and add or delete key/value pairs. One can also iterate through all keys, all values, or all key/value pairs. (Actually the implementation in OTI Smalltalk uses LookupTable, but this has the same functionality.)

The solver itself is represented as an instance of ClSimplexSolver, with public message protocol as described above. There is more on the implementation of this class in Subsection 4.3.

4.2.1 Variables

ClAbstractVariable and its subclasses represent various kinds of constrained variables. ClAbstract-Variable is an abstract class, that is, it is just used as a superclass of other classes; one does't 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 was used for debugging — a final version might not need this.)

Instances of ClVariable 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 don't 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 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.

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

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

For isExternal, instances of ClVariable return true and everyone else returns 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 returns true and everyone else returns 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.

So variables don't hold state, except for a name for debugging, and a value for instances of ClVariable — mostly their significance is just their identity. The only other messages that variables understand are some messages to ClVariable for creating constraints — see Subsection 4.2.5.

4.2.2 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 isn't 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. This representation was convenient in Smalltalk given the built-in

class dictionary, and allows one to find the coefficient for a given variable without searching. It has some space overhead for the dictionary. An alternative representation would be to use a linked list for the coefficients and variables — with this representation one would need to search the list for a given variable, but the representation would be more compact. If expressions typically had only a small number of non-zero coefficients this representation may be preferable.

Linear expressions understand a large number of messages. Some of these are for constraint creation (see Section 4.2.5). 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.

4.2.3 Constraints

There is an abstract class Constraint 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 weight of the constraint, or nil if it doesn't have a weight. (Weights are only relevant for the weighted-sum-better comparator, not for locally-error-better.)

Constraints understand various message 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 has an instance variable expression, which will hold an instance of ClLinearExpression. It has two concrete subclasses. An instance of ClLinearEquation represents the linear equality constraint

expression = 0.

An instance of ClLinearInequality represents the constraint

expression $\geq$ 0.

The other part of the hierarchy is for edit and stay constraints (both of which are represented explicitly in the current implementation). ClEditOrStayConstraint has an instance variable variable, which is the variable with the edit or stay. Otherwise all they do is respond appropriately to the messages isEditConstraint and isStayConstraint.

4.2.4 Possible Revisions to Constraint Representation

In contrast to the current Cassowary implementation described above, the QOCA implementation doesn't represent edits or stays explicitly. Rather, each variable has a preferred value and a weight, in addition to a current value. Given the special treatment of edit constraints in Cassowary (and since there are typically only one or two of them) it is probably worth continuing to represent them explicitly. However, since many variables have a stay it may be worth representing stay constraints implicitly. Then every constrainable variable would automatically be given an implicit stay constraint of a given strength, which could be a special null strength if no stay were desired. This strength would be stored as an instance variable of ClVariable. This would only allow at most one stay per variable, which is the typical situation — if multiple stays were needed for some reason this could be simulated using additional variables and equality constraints.

This 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.

4.2.5 Constraint Creation

This subsection describes a mechanism to allow constraints to be defined easily by programmers. Unlike the material described in other sections, this may not be easily implementable in languages other than 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 $3x+5 \leq y$. This works as follows. The number 3 gets the message * x. Since x isn't 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 isn't 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 $3x+5-y$, 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 since 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.)

4.2.6 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 — see below — but we wouldn't expect to do so frequently.)

The strengths are currently defined as follows.

required Required constraints must be satisfied. This strength is used for most programmer-defined constraints.

strong This strength is used for edit constraints.

medium Currently unused.

weak This strength is used for stay constraints.

These are represented as 4 instances of ClStrength.

The other relevant class is ClSymbolicWeight. As mentioned in Section 2.5, 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, rather than real numbers as weights we use symbolic weights and

a lexicographic ordering, 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, as follows:

+ w
: w is also a symbolic weight. Return the result of adding self to w.

- 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
: w is a symbolic weight. Return true if self is less than or equal to n.

>= n
: Similarly.

< n
: Similarly.

> n
: Similarly.

= n
: Similarly.

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).

These messages let the user of symbolic weights (i.e. the solver) use them just like numbers in expressions.

Finally, 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 $(0.0, 0.0, 1.0)$.

4.3 ClSimplexSolver Implementation

Here are the instance variables of ClSimplexSolver.

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 many 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

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

infeasibleRows

Return a set of basic variables that have infeasible rows. (This 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

An array of plus error variables (instances of ClSlackVariable) for the stay constraints. The corresponding negative error variable must have the same index in stayMinusErrorVars.

stayMinusErrorVars

See stayPlusErrorVars.

editPlusErrorVars

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

editMinusErrorVars

See 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 reset).

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, since the corresponding error variables must be deleted from the objective function.

slackCounter

Used for debugging. An integer used to generate names for slack variables, which are useful when printing out expressions. (Thus we get slack variables named s1, s2, etc.)

artificialCounter
: Similar to slackCounter but for artificial variables.

dummyCounter
: Similar to slackCounter but for dummy variables (ie. marker variables for required equality constraints).

4.3.1 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
- determine whether a variable is 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 a row
- 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 is a bit tricky, and so the solver actually accesses the rows and columns only via the following interface, to avoid getting the two representations out of sync. (This isn't really clean — what would in retrospect be better is to have a separate class ClTableau that holds the rows and columns and that supports this interface.)

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

noteAddedVariable: var subject: subject
: var and subject are both ClAbstractVariables. Variable var has been added to the linear expression for subject. Update the column cross indices.

noteRemovedVariable: var subject: subject
: var and subject are both ClAbstractVariables. Variable var has been removed from the linear expression for subject. Update the column cross indices.

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

removeRow: var

Remove the basic variable var from the tableau. Since var is basic, there should be a row var=expr. Remove this row, and also update the column cross indices.

substituteOut: var expr: expr

var is a ClAbstractVariable and expr is a ClLinearExpression. Replace all occurences of var with expr and update the column cross indices.

4.3.2 Adding a Constraint

Section 2.3 discussed how to add constraints incrementally. For efficiency we should avoid using an artificial variable if possible. 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 expr (which is an instance of ClLinearExpression). If necessary, normalize expr by multiplying by -1 so that its constant part is non-negative. We are adding the constraint expr=0 to the tableau. To do this we want to pick a variable in expr to be the subject of an equation, so that we can add the row var=expr2, where expr2 is the result of solving expr=0 for var.

- If expr contains any unrestricted variables, we must choose an unrestricted variable as the subject.
- If the subject is new to the solver, we won't have to do any substitutions, so we prefer new variables to ones that are currently noted as parametric.
- If expr contains only restricted variables, if there is a (restricted) variable in expr 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, since 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 is good because a common operation is adding a non-required edit.

4.3.3 Removing a Constraint

Here are a few additional remarks in addition to the material presented in Section 2.4.

First, 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. Instead, 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. (s is an instance of ClSymbolicWeight and w is a float.)

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).

4.4 Omissions

The solver should implement Bland's anti-cycling rule, but it doesn't at the moment. Adding this should be straightforward.

4.5 Comparator Details

Our implementation of Cassowary favors solutions that satisfies some of the constraints completely, rather than ones that partially satisfy e.g. each of two conflicting equalities. These are still legitimate locally-error-better solutions. Cassowary's behaviour 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. (And of course Cassowary behaves this way because simplex does.)

Such solutions are also produced by greedy constraint satisfaction algorithms, such as local propagation algorithms like DeltaBlue and Indigo, since 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, which has not yet been resolved in an entirely clean way. One of the public methods for Cassowary is addPointStays: points, as discussed in Subsection 4.1. 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. (In the earlier implementations of Cassowary this happened, and indeed it looked strange — so this claim has been verified empirically.)

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

I had some trouble coming up with an example where it would give a bad answer — here is a kind of 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 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 e.g. $\{x \mapsto 3.8\}$ — although this hypothesis hasn't been tested. Another relevant question is whether users prefer any of these solutions over others (for a given application domain).

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 actually been a bit sloppy 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 specificially 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 ($p1.x$ *stay* & $p1.y$ *stay*)
weak ($p2.x$ *stay* & $p2.y$ *stay*)

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. This produces the desired result.

Note that 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 and hence satisfy the stay on the x component of one point and the y component of another.

Note also that all of this is not a problem for QOCA — its least-squares-better comparator distributes the error to the x and y components of all the points with stays of the same strength.

5 QOCA: Quadratic Optimization

Another useful way of comparing solutions to constraint hierarchies is least-squares-better, in which case we are interested in solving optimization problems of the form, referred to as QP:

minimize f subject to C
where $f = \sum_{i=1}^{n} w_i(x_i - d_i)^2$

The variables are $x_1, \ldots, x_n$, and C is the set of required constraints. The desired value for variable x_i is d_i, and the "weight" associated with that desire (which should reflect the hierarchy) is w_i.

This problem is a type of *quadratic programming* in which a quadratic optimization function is minimized with respect to a set of linear arithmetic equality and inequality constraints. In particular, since the optimization function is a sum of squares, the problem is an example of *convex* quadratic programming, meaning that the local minimum is also the global minimum. This is fortunate, since convex quadratic programming has been well-studied and efficient methods for solving these problems are well-known in the operations research community. Here we will present two methods. The first is a variant of the simplex algorithm introduced earlier, while the second, based on "active sets," is the method of choice for medium scale problems consisting of up to 1000 variables and constraints.

5.1 Linear Complementary Pivoting

Arguably the simplest approach to solving convex quadratic problems is a simple modification of the simplex algorithm that finds the local optimum of a quadratic problem, which since the problem is convex, is the global optimimum.

Now, a solution is a local minimum if in every direction either the optimization value increases or the region becomes infeasible. The information about infeasibility is captured by the constraints in the original problem (called the *primal problem*). Information about how the optimization function decreases is captured in the so-called *dual problem* which is obtained by looking at the derivative of the optimization function. The idea is therefore to combine the primal and dual problems and solve these together. Any solution to their combination will be a feasible optimal solution for the original problem. The point about a quadratic problem is that the derivative of a quadratic optimization function is linear. Thus both the dual and the primal problem consist of linear arithmetic constraints and so a variant of the simplex can be used to solve their conjunction. Let the constraints in the primal problem be in basic feasible solved form:

$$PP: \qquad \bigwedge_{i=1}^{n} x_i = b_i - \sum_{j=1}^{m} a_{ij} y_j$$

where $x_1, ..., x_n$ are the basic variables and $y_1, ..., y_m$ are the parameters and let the function to be minimized be O and assume that basic variables have been eliminated from O. Then the dual problem is

$$DP: \qquad \bigwedge_{j=1}^{m} t_j = \frac{\partial O}{\partial y_j} + \sum_{i=1}^{n} a_{ij} z_i$$

where $z_1, ..., z_n \geq 0$ are the dual variables (one for each equation in the primal problem) and $t_1, ..., t_m \geq 0$ are the dual slack variables.

The combined problem CP is the conjunction of the dual and primal problem plus the constraints that for all i and j, $x_i \times z_i = 0$ and $y_j \times t_j = 0$. Note that the last constraints mean that in the combined problem every variable has a *complementary* variable which is not allowed to be positive if it is.

For example, imagine that our primal problem is the constraints from before:

$$\begin{array}{rcrrr}
x_m & = & 50 & +\frac{1}{2}x_l & -\frac{1}{2}s_4 \\
s_1 & = & 90 & -x_l & -s_4 \\
s_2 & = & 100 & -x_l & \\
s_3 & = & 95 & -\frac{1}{2}x_l & +\frac{1}{2}s_4 \\
x_r & = & 100 & & -s_4
\end{array}$$

and that we have the weak stay constraints $x_l = 10$, $x_m = 50$ and $x_r = 90$. The optimization problem is therefore to minimize

$$(x_l - 10)^2 + (x_m - 50)^2 + (x_r - 90)^2.$$

Eliminating basic variables from this function gives

$$O = (x_l - 10)^2 + (\frac{1}{2}x_l - \frac{1}{2}s_4)^2 + (10 - s_4)^2.$$

Now

$$\frac{\partial O}{\partial x_l} = 2(x_l - 10) + (2 \times \frac{1}{2})(\frac{1}{2}x_l - \frac{1}{2}s_4) = \frac{5}{2}x_l - \frac{1}{2}s_4 - 20$$

and

$$\frac{\partial O}{\partial s_4} = (2 \times -\frac{1}{2})(\frac{1}{2}x_l - \frac{1}{2}s_4) - 2(10 - s_4) = -\frac{1}{2}x_l + \frac{5}{2}s_4 - 20.$$

The dual problem is therefore

$$\begin{array}{rcrrrrrrr}
t_1 & = & -20 & +\frac{5}{2}x_l & -\frac{1}{2}s_4 & +\frac{1}{2}z_1 & -z_2 & -z_3 & -\frac{1}{2}z_4 & \\
t_2 & = & -20 & +-\frac{1}{2}x_l & +\frac{5}{2}s_4 & -\frac{1}{2}z_1 & -z_2 & & +\frac{1}{2}z_4 & -z_5
\end{array}$$

Putting the primal and dual together we obtain the problem:

$$\begin{array}{rcrrrrrrr}
x_m & = & 50 & +\frac{1}{2}x_l & -\frac{1}{2}s_4 & & & & \\
s_1 & = & 90 & -x_l & -s_4 & & & & \\
s_2 & = & 100 & -x_l & & & & & \\
s_3 & = & 95 & -\frac{1}{2}x_l & +\frac{1}{2}s_4 & & & & \\
x_r & = & 100 & & -s_4 & & & & \\
t_1 & = & -20 & +\frac{5}{2}x_l & -\frac{1}{2}s_4 & +\frac{1}{2}z_1 & -z_2 & -z_3 & -\frac{1}{2}z_4 & \\
t_2 & = & -20 & +-\frac{1}{2}x_l & +\frac{5}{2}s_4 & -\frac{1}{2}z_1 & -z_2 & & +\frac{1}{2}z_4 & -z_5
\end{array}$$

where the variables in the problem and their complements are given by:

$$\begin{array}{ll}
x_l \leftrightarrow t_1 & x_m \leftrightarrow z_1 \\
x_r \leftrightarrow z_5 & s_1 \leftrightarrow z_2 \\
s_2 \leftrightarrow z_3 & s_3 \leftrightarrow z_4 \\
s_4 \leftrightarrow t_2 &
\end{array}$$

A consequence of the way the dual is constructed is that a variable and its complement cannot both be basic, meaning that the solution read from the basic feasible solved form automatically satisfies the complementary condition.

Unfortunately the complete problem is not in basic feasible form. To transform it into basic feasible form we add an artificial variable v to every equation giving the equations:

$$
\begin{array}{rcrrrrrrrrr}
x_m & = & 50 & +v & +\frac{1}{2}x_l & -\frac{1}{2}s_4 & & & & & \\
s_1 & = & 90 & +v & -x_l & -s_4 & & & & & \\
s_2 & = & 100 & +v & -x_l & & & & & & \\
s_3 & = & 95 & +v & -\frac{1}{2}x_l & +\frac{1}{2}s_4 & & & & & \\
x_r & = & 100 & +v & & -s_4 & & & & & \\
t_1 & = & -20 & +v & +\frac{5}{2}x_l & -\frac{1}{2}s_4 & +\frac{1}{2}z_1 & -z_2 & -z_3 & -\frac{1}{2}z_4 & \\
t_2 & = & -20 & +v & -\frac{1}{2}x_l & +\frac{5}{2}s_4 & -\frac{1}{2}z_1 & -z_2 & & +\frac{1}{2}z_4 & -z_5
\end{array}
$$

We now pivot on the row with the largest negative constant. In this case either the row with basic variable t_1 or that with t_2. Say we arbitrarily choose t_2 then we obtain:

$$
\begin{array}{rcrrrrrrrrr}
x_m & = & 70 & +t_2 & +x_l & -3s_4 & +\frac{1}{2}z_1 & +z_2 & & -\frac{1}{2}z_4 & +z_5 \\
s_1 & = & 110 & +t_2 & -\frac{1}{2}x_l & +\frac{3}{2}s_4 & +\frac{1}{2}z_1 & +z_2 & & -\frac{1}{2}z_4 & +z_5 \\
s_2 & = & 120 & +t_2 & -\frac{1}{2}x_l & -\frac{5}{2}s_4 & +\frac{1}{2}z_1 & +z_2 & & -\frac{1}{2}z_4 & +z_5 \\
s_3 & = & 115 & +t_2 & & -2s_4 & +\frac{1}{2}z_1 & +z_2 & & -\frac{1}{2}z_4 & +z_5 \\
x_r & = & 120 & +t_2 & +\frac{1}{2}x_l & -\frac{7}{2}s_4 & +\frac{1}{2}z_1 & +z_2 & & -\frac{1}{2}z_4 & +z_5 \\
t_1 & = & 0 & +t_2 & +3x_l & -3s_4 & +z_1 & & -z_3 & -z_4 & +z_5 \\
v & = & -20 & +t_2 & +\frac{1}{2}x_l & -\frac{5}{2}s_4 & +\frac{1}{2}z_1 & +z_2 & & -\frac{1}{2}z_4 & +z_5
\end{array}
$$

We have now obtained a solved form which is basic feasible and which satisfies the complementary conditions. The only problem is that the artificial variable v is in the basis. We continue pivoting until v leaves the basis. At each pivot we choose to make basic the variable which is complementary to the variable which just left the basis. Thus in the next stage we would choose to move s_4 into the basis since this is the complement of t_2. We use the standard simplex row selection rule to determine which variable to move out of the basis. This ensures that feasibility is maintained. Because we only move a variable into the basis once its complementary variable has been moved out of the basis this means that the solution corresponding to any of the solved forms will satisfy the complementary conditions.

In analogy with the incremental simplex algorithm we can modify this algorithm so that it is incremental for resolving. Changing the desired variable values only changes the constants in the solved form. There are two possibilities. If the solved form remains feasible, then we just read the new solution directly from the solved form. Otherwise, the solved form is now infeasible. In this case we proceed as above, first introducing an artificial variable v and making the solved form feasible, then pivoting until v leaves the basis.

We are currently exploring an implementation based on complementary pivoting.

5.2 Active Set Method

Our current implementation of QOCA uses the *active set method* [7] to solve the convex quadratic programming problem. This method is an iterative technique for solving constrained optimization problems with inequality constraints. It is reasonably robust and quite fast, and is the method of choice for medium scale problems consisting of up to 1000 variables and constraints.

The key idea behind the algorithm is to solve a sequence of constrained optimization problems O_0, ..., O_t ($t \geq 0$). Each problem minimizes f with respect to a set of equality constraints, $\mathcal{A}$, called the *active set*. The active set consists of the original equality constraints plus those inequality

constraints that are "tight," in other words, those inequalities that are currently required to be satisfied as equalities. The other inequalities are ignored for the moment.

Essentially, each optimization problem O_i can be treated as an unconstrained quadratic optimization problem, denoted by U_i. To obtain U_i, we rewrite the equality constraints in O_i in basic feasible solved form, and then eliminate all basic variables in the objective function f. The optimal solution is the point at which all of the partial derivatives of f equal zero. The problem U_i can be solved easily, since we are dealing with a convex quadratic function f and so its derivatives are linear. As a result, to solve U_i we need only solve a system of linear equations over unrestricted variables.

In more detail, in the active set method, we assume at each stage that a feasible initial guess $\mathbf{x}_0 = (x_1, \cdots x_n)^T$ is available, as well as the corresponding active set $\mathcal{A}$. Assume that we have just solved the optimization problem O_0, and let its solution be $\mathbf{x}_0^*$. We face the following two possibilities when determining the new approximate solution $\mathbf{x}_1$.

1. $\mathbf{x}_0^*$ is feasible with respect to the constraints in O_0 but it violates some inequality constraints in QP that are not in the current active set $\mathcal{A}$. In this case, a scalar $\alpha \in [0, 1]$ is selected, such that it is as large as possible and the point $\mathbf{x}_0 + \alpha(\mathbf{x}_0^* - \mathbf{x}_0)$ is feasible. This point is taken as the new approximate solution $\mathbf{x}_1$, and the violated constraints are added to the active set, giving rise to a new optimization problem O_1.
2. $\mathbf{x}_0^*$ is feasible with respect to the original problem QP. It is directly taken as the new approximate solution $\mathbf{x}_1$ and we test to see it is also optimal QP. This requires us to check if there exists a direction $\mathbf{s}$ at $\mathbf{x}_1$, such that a feasible incremental step along $\mathbf{s}$ reduces f. If such direction $\mathbf{s}$ exists, then one constraint is taken out of the active set $\mathcal{A}$ to generate the direction $\mathbf{s}$, which results in a new optimization problem O_1. If no such direction exists we are finished since $\mathbf{x}_1$ is both feasible and optimal.

If the active set is modified, the whole process is repeated until the optimal solution is reached.

Note that the active set method is closely related to the simplex method. Those inequalities whose slack variables are not basic are in the active set, while those whose slack variables are basic are not. Pivoting corresponds to moving one inequality out of the active set and replacing it by another.

Consider our working example with the weak constraints that $x_m = 50$, $x_l = 30$ and $x_r = 70$. This gives rise to the minimization problem QP_1 :

minimize $f_1 = (x_m - 50)^2 + (x_l - 30)^2 + (x_r - 70)^2$ subject to

$$\begin{array}{lrrrcr} (1) & 2x_m & -x_l & -x_r & = & 0 \\ (2) & & -x_l & +x_r & \geq & 10 \\ (3) & & & -x_r & \geq & -100 \\ (4) & & x_l & & \geq & 0 \end{array}$$

Although it is obvious that $x_m = 50, x_l = 30, x_r = 70$ or $\mathbf{x}^* = (50, 30, 70)^T$ is the optimal solution, it is still instructive to see how the active set method computes this. The initial guess and active set are read from the augmented simplex form tableux. We start with an initial guess $x_m = 50, x_l = 0, x_r = 100$, i.e. $\mathbf{x}_0 = (50, 0, 100)^T$, and constraints 1, 3 and 4 are active. Thus $\mathcal{A}_0^{(1)} = \{1, 3, 4\}$ is the initial active set. The equality constrained optimization problem $O_0^{(1)}$ is therefore

minimize f_1 subject to

$$\begin{array}{rrrcr} 2x_m & -x_l & -x_r & = & 0 \\ & & -x_r & = & -100 \\ & x_l & & = & 0 \end{array}$$

The problem $O_0^{(1)}$ has only one feasible solution $x_m = 50, x_l = 0, x_r = 100$, so it is also the optimal solution, denoted by $\mathbf{x}_0^*$. Next we check if $\mathbf{x}_0^*$ is the optimal solution to the problem QP_1. Constraint 4 forces x_l to take the value 0 in $\mathbf{x}_0^*$. However, the value of the objective function f_1 can be reduced if x_l is increased. Thus the 4th constraint $\mathbf{x}_l \geq 0$ can be moved out of the active set in order to further reduce the value of f_1. This gives $\mathbf{x}_1 = \mathbf{x}_0^*$ as the new approximate solution, $\mathcal{A}_1^{(1)} = \{1, 3\}$ as the active set and the optimization problem $O_1^{(1)}$ as

minimize f_1 subject to

$$\begin{array}{rrrcr} 2x_m & -x_l & -x_r & = & 0 \\ & & -x_r & = & -100 \end{array}$$

To solve $O_1^{(1)}$, we rewrite the constraints in $O_1^{(1)}$ to a basic feasible solved form $x_r = 100 \wedge x_l = 2x_m - 100$, and then eliminate basic variables in the function f_1. This results in the following unconstrained optimization problem

minimize $(x_m - 50)^2 + (2x_m - 100 - 30)^2 + (100 - 70)^2$

Setting the derivative to be zero we obtain

$$2(x_m - 50) + 2 \times 2(2x_m - 130) = 0.$$

Solving this together with the constraint in $O_1^{(1)}$, the optimal solution of $O_1^{(1)}$ is found to be $\mathbf{x}_1^* = (62, 24, 100)^T$. It is easy to verify that $\mathbf{x}_1^*$ is still feasible. Similarly to the case for $\mathbf{x}_0^*$, in $\mathbf{x}_1^*$ x_r is forced to take the value 100 because of the 3rd constraint, yet the function value f_1 can be reduced if x_r is decreased. So the 3rd constraint $-x_r \geq -100$ is moved out of the active set. We now have the new approximate solution $\mathbf{x}_2 = \mathbf{x}_1^*$, the active set $\mathcal{A}_2^{(1)} = \{1\}$ and the optimization problem $O_2^{(1)}$:

$$\text{minimize } f_1 \text{ subject to } 2x_m - x_l - x_r = 0.$$

To solve this problem, we repeat the same procedure as for solving $O_1^{(1)}$. The solution to this problem satisfies the equations:

$$\begin{array}{lcl} 2(x_m - 50) & + & 2 \times 2(2x_m - x_l - 70) \quad = \quad 0 \\ 2(x_l - 30) & + & 2(2x_m - x_l - 70) \quad = \quad 0 \end{array} \tag{1}$$

These together with the constraint in $O_2^{(1)}$ have the solution $\mathbf{x}^* = (50, 30, 70)^T$. This is the optimal solution to $O_2^{(1)}$ and is also the optimal solution to the original problem QP_1.

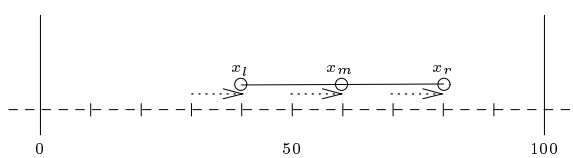

Figure 5: Resolving the constraints using QOCA

Now imagine that we have started to manipulate the diagram. We have the weak constraints that $x_l = 30$ and $x_r = 70$ and the strong constraint that $x_m = 60$. Reflecting this, we change the first term in the function f_1 to be $1000(x_m - 60)^2$, denote it as f_2 and the corresponding optimization

problem as QP_2. Starting from $\mathbf{x}_0 = (50, 30, 70)^T$, which is the optimal solution to QP_1, an equality constrained problem $O_0^{(2)}$ is formed. $O_0^{(2)}$ is the same as $O_2^{(1)}$, except that they have different objective functions. The solution to $O_0^{(2)}$ satisfies similar linear equations to those of (1). These can be obtained by replacing the term $2(x_m - 50)$ in the first equation of (1) by $1000(x_m - 60)$ reflecting the change in the objective function. A solved form for these equations is

$$\begin{array}{lclcl} x_m & = & \frac{500}{501} \times 60 & + & \frac{50}{501} \\ x_l & = & x_m & - & 20 \end{array} \tag{2}$$

which leads to the optimal solution for both $O_0^{(2)}$ and QP_2 as $x_m = 59.98, x_l = 39.98, x_r = 79.98$.

Note that the exact least-squares-better solution is actually $x_m = 60, x_l = 40, x_r = 80$. With quadratic optimization the strong constraints don't completely dominate the weak ones in the computed solution. However, by choosing a suitably large constant we found a solution that *is* least-squares-better to under a one-pixel resolution, so that the deviation from a least-squares-better solution would not be visible in an interactive system.

In practice, this appears to work well. However, in principle it is possible to solve the problem iteratively by computing the solution to a sequence of quadratic optimization problems in which the constants grow by an order of magnitude and stopping when the solution is sufficiently close to the limit. It might also be possible to solve the problem by using symbolic values, however, this is more difficult than in the Cassowary algorithm because of more complex manipulation.

To modify the active set method so that it is incremental for resolving, we observe that changing the desired variable values only changes the optimization function f. Thus we can reuse the active set from the last resolve and reoptimize with respect to this. In most cases the active set does not change, and so we are done. Otherwise we proceed as above.

For example, if we now move x_m from 60 to 90, we change the objective function again, but need only change the desired values and can keep the weights the same as they are in f_2, e.g. in the new objective function f_3, the variable x_m has a new desired value 90. The corresponding optimization problem is referred to as QP_3. To solve this problem, the *resolve* procedure makes use of the information from the previous solve QP_2, while applying the active set method to QP_3. When resolving, it is important to notice that, if we start from the solution for the previous problem QP_2, i.e. $\mathbf{x}_0 = (59.98, 39.98, 79.98)^T$, then the solution to the corresponding equality constrained problem $O_0^{(3)}$,

$$\text{minimize } f_3 \text{ subject to } 2x_m - x_l - x_r = 0,$$

can be easily obtained. In fact, one can just replace the desired value 60 for x_m in (2) by its new desired value 90, which leads to the optimal solution to $O_3^{(0)}$ as $\mathbf{x}_0^* = (89.9202, 69.9202, 109.9202)^T$. If the desired value does not change too much, it is quite likely that $\mathbf{x}_0^*$ is also optimal for QP_3. Unfortunately, this is not the case for this example, since $\mathbf{x}_0^*$ violates the 3rd constraint $-x_r \geq -100$. Choosing $\alpha \in [0, 1]$ to be as big as possible while still ensuring that $\mathbf{x}_1 = \mathbf{x}_0 + \alpha(\mathbf{x}_0^* - \mathbf{x}_0)$ is feasible, we have $\alpha = 0.6687$ and $\mathbf{x}_1 = \mathbf{x}_0 + \alpha(\mathbf{x}_0^* - \mathbf{x}_0)$ as the new approximate solution, at which the 3rd constraint becomes active. By solving the corresponding equality constrained problem $Q_1^{(3)}$,

$$\text{minimize } f_3 \text{ subject to } 2x_m - x_l - x_r = 0,\ -x_r = -100,$$

the optimal solution to QP_3 is found to be $x_m = 89.9003$, $x_l = 79.8007$, $x_r = 100$.

Figure 5 shows the effect of moving the horizontal line with the least squares comparator. With this comparator the line moves right maintaining the same length until it hits the right boundary,

at which point it starts to compress. This contrasts with the behaviour of the locally-error-better comparator in which the line grew until it bumped against the side.

The actual implementation of QOCA is rather more complex than this example suggests and the reader is referred to [7] for more details.

6 Empirical Evaluation

Both algorithms have been implemented and tested.

Our algorithms for incremental addition and deletion of equality and inequality constraints and for solving and resolving for the least-square comparator using the QOCA algorithm have been implemented as part of the QOCA C++ constraint solving toolkit. The results are very satisfactory. For a test problem with 300 constraints and 300 variables, adding a constraint takes on average 1.5 msec, deleting a constraint 1.6 msec, the initial solve 12 msec, and subsequent resolving as the point moves 4.5 msec. For a larger problem with 900 constraints and variables, adding a constraint takes on average 9.7 msec, deleting a constraint 17 msec, the initial solve 120 msec, and subsequent resolving as the point moves 67 msec. These tests were run a sun4m sparc, running SunOS 5.4.

Cassowary has been initially implemented in Smalltalk. Running the Smalltalk implementation of Cassowary on the same problems, for the 300 constraint 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.) For the 900 constraint problem, adding a constraint takes on average 98 msec (again including the initial solve), 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.

We have recently reimplemented Cassowary in Java, but haven't done any performance measurements yet.

As these measurements are for implementations in different languages, running on different machines, they should not be viewed as any kind of head-to-head comparison. Nevertheless, they indicate that both algorithms are eminently practical for use with interactive graphical applications.

The QOCA toolkit has been employed in a number of applications. The first application is part of an intelligent pen and paper interface that contains a parser to incrementally parse diagrams drawn by the user using a stylus, and that has a diagram editor that respects the semantics of the diagram by preserving the constraints recognized in the parsing process. QOCA is used for both error correction in parsing and for diagram manipulation in the editor [6]. A second QOCA application is for layout of trees and graphs in the presence of arbitrary linear arithmetic constraints and with suggested placements for some nodes [8].

A Cassowary application currently being developed using the Java implementation is a web authoring tool [5], in which the appearance of a page is determined by constraints from both the web author and the viewer.

Acknowledgments

This project has been funded in part by the National Science Foundation under Grants IRI-9302249 and CCR-9402551 and in part by Object Technology International. Alan Borning's visit to Monash University and the University of Melbourne was sponsored in part by the Australian-American Educational Foundation (Fulbright Commission).

References

[1] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94 Conference Proceedings*, pages 23–32. ACM, 1994.

[2] Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology*, pages 129–136, Seattle, November 1996.

[3] Alan Borning and Bjorn Freeman-Benson. 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*, pages 624–628, Cassis, France, September 1995.

[4] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[5] Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of ACM MULTIMEDIA'97*, November 1997.

[6] S.S. Chok and K. Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *IEEE Symposium on Visual Languages*, pages 242–250, 1995.

[7] Roger Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, Chichester, New York, Brisbane, Toronto, Singapore, 1987.

[8] W. He and K. Marriott. Constrained graph layout. In *Graph Drawing '96*, volume 1190 of *LNCS*, pages 217–232. Springer-Verlag, 1996.

[9] Richard Helm, Tien Huynh, Catherine Lassez, and Kim Marriott. A linear constraint technology for interactive graphic systems. In *Graphics Interface '92*, pages 301–309, 1992.

[10] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Proceedings of the Third Eurographics Workshop on Object-oriented Graphics*, Champery, Switzerland, October 1992.

[11] Hiroshi Hosobe, Satoshi Matsuoka, and Akinori Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, Boston, August 1996.

[12] Scott Hudson and Ian Smith. SubArctic UI toolkit user's manual. Technical report, College of Computing, Georgia Institute of Technology, 1996.

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

[14] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[15] Kim Marriott and Peter Stuckey. *Introduction to Constraint Logic Programming.* Mit Press, 1997. In preparation.

[16] Brad A. Myers. The Amulet user interface development environment. In *CHI'96 Conference Companion: Human Factors in Computing Systems*, Vancouver, B.C., April 1996. ACM SIGCHI.

[17] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, second edition, 1989.

[18] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[19] Ivan Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.

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