Solving Linear Arithmetic Constraints for User Interface Applications

Alan Borning and Kim Marriott*
Department of Computer Science
Monash University
Clayton, Victoria 3168, AUSTRALIA
{borning,marriott}@cs.monash.edu.au

Peter Stuckey and Yi Xiao
Department of Computer Science;
Department of Mathematics & Statistics
University of Melbourne
Parkville, Victoria 3052, AUSTRALIA
pjs@cs.mu.oz.au; yxiao@maths.mu.oz.au

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.

KEYWORDS: Linear constraints, inequality constraints, simplex algorithm

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 giving an error. A third use is to balance conflicting desires, for example in laying out a graph.

*Permanent address: Department of Computer Science & Engineering, University of Washington, Box 352350, Seattle, WA 98195, USA

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.

As described in [2], it is important to use a metric rather than a predicate comparator for inequality constraints. Thus, plausible comparators for use with linear equality and inequality constraints are *locally-error-better*, *weighted-sum-better*, and *least-squares-better*. 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. In UI 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 resatisfy 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 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 4 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 in Smalltalk and QOCA in C++. They perform surprisingly well, and a summary of our results is given in Section 5. 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. A companion technical report [6] gives additional details for both algorithms.

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 [20]. Most of the current systems use one-way constraints (e.g. [13, 17]), or local propagation algorithms for acyclic collections of multi-way constraints (e.g. [19, 21]). Indigo [2] handles acyclic collections of inequality constraints, but not cycles (simultaneous equations and inequalities). UI systems that handle simultaneous linear equations include DETAIL [12] 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 [10] and [11]. These earlier descriptions, however, do not give any details of the algorithm, although the incremental deletion algorithm is described in [14]. The current implementation is much improved, in particular through the use of the active set method described in Section 4.1.

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($\mathcal{R}$) [15] include efficient solvers for linear equalities and inequalities. (See [16] 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*, [18, page 424]

We now describe an incremental version of the simplex algorithm, adapted to the task at hand. In the description we use a running example, illustrated by the diagram in Figure 1.

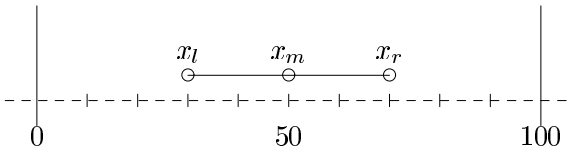

Figure 1: Simple constrained picture

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. 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 $\bigwedge\{x \geq 0 \mid x \text{ is a variable in } C_S\}$, 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 [16], and we have adopted it here. Essentially it uses *two* tableaux rather than one. Equations containing at least one unrestricted variable will be placed in C_U, the unrestricted variable tableau while C_S, the simplex tableau, contains those equations in which all variables are 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.

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 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 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 the second phase 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.

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

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

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. This is achieved by choosing 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 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.

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.

As an example, suppose we wish to ensure the additional constraint 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 the first phase 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 an non-negative 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.

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" nonnegative 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. (For simplicity we didn't include this variable in the tableaux presented earlier.)

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, determining the most restrictive equation, and using that equation to pivot s_1 into the basis. (See [6] for details.) We then remove the row. Here 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

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

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 [6]. However, QOCA is not able to 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.

Translating the constraints strong $x_m = 50$, weak $x_l = 30$, and weak $x_r = 60$ that 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

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

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

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 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 this row 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

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

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

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 2.

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

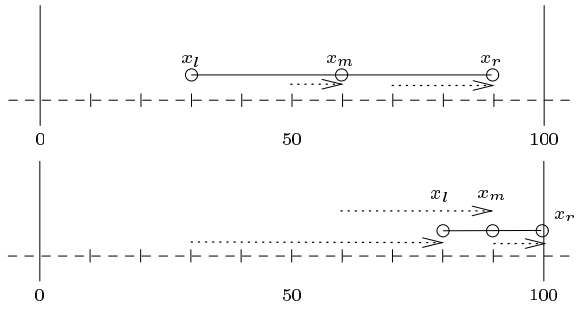

Figure 2: Resolving the constraints

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

x_m	$=$	90	$+\delta^+_{x_m}$	$-\delta^-_{x_m}$			
x_r	$=$	150	$+2\delta^+_{x_m}$	$-2\delta^-_{x_m}$	$-\delta^+_{x_l}$	$+\delta^-_{x_l}$	
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	$+2\delta^+_{x_m}$	$-2\delta^-_{x_m}$	$-\delta^+_{x_l}$	$+\delta^-_{x_l}$	$+\delta^-_{x_r}$

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. 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, replacing y_j by

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

is performed as in the (primal) simplex algorithm.

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

x_m	$=$	90		$+\delta^+_{x_m}$	$-\delta^-_{x_m}$		
x_r	$=$	100	$-s_2$				
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	$-s_2$				$+\delta^-_{x_r}$

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 2.

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 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 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 following 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.

4.1 Active Set Method

Our implementation of QOCA uses the *active set method* [8] 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 $O_0, ..., O_t$. 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 unconstrained 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.

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

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

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

$$\text{minimize } (x_m-50)^2+(2x_m-100-30)^2+(100-70)^2$$

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:

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

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.

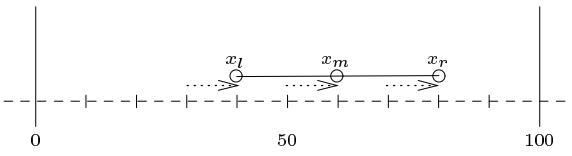

Figure 3: Resolving the constraints using QOCA

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

$$\begin{array}{rcrcl} x_m & = & \frac{500}{501} \times 60 & + & \frac{50}{501} \\ x_l & = & x_m & - & 20 \end{array} \quad (2)$$

which leads to the optimal solution for both $O_0^{(2)}$ and QP_2 as $x_m = 59.98, x_l = 39.98, x_r = 79.98$. (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. (See [6] for more on this issue.)

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 3 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.

5 EMPIRICAL EVALUATION

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.

Running 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.

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 [7]. 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 [9]. A Cassowary application currently being developed 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 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. D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94*, pages 23–32.
2. A. Borning, R. Anderson, and B. Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *UIST'96*, pages 129–136, Seattle, Nov 1996.
3. A. Borning and B. Freeman-Benson. The OTI constraint solver: A constraint library for constructing interactive graphical user interfaces. In *Proc. Constraint Programming'95*, Springer-Verlag LNCS Vol 910, Sep 1995.
4. A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, Sep 1992.
5. A. Borning, R. Lin, and K. Marriott. Constraints for the web. In *Proc. ACM MULTIMEDIA'97*, Nov 1997. To appear.
6. A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications: Algorithm details. Tech report 97-06-01, Dept. Computer Science & Engr, Univ of Washington, Seattle, WA, July 1997.
7. 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.
8. R. Fletcher. *Practical Methods of Optimization*. Wiley, 1987.
9. W. He and K. Marriott. Constrained graph layout. In *Graph Drawing '96*, Springer-Verlag LNCS Vol 1190, pages 217–232.
10. R. Helm, T. Huynh, C. Lassez, and K. Marriott. A linear constraint technology for interactive graphic systems. In *Graphics Interface '92*, pages 301–309, 1992.
11. R. Helm, T. Huynh, K. Marriott, and J. Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Proc. Third Eurographics Workshop on Object-oriented Graphics*, Champery, Switzerland, Oct 1992.
12. H. Hosobe, S. Matsuoka, and A. Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *Proc. Constraint Programming'96*, Springer-Verlag LNCS Vol 1118, Aug 1996.
13. S.E. Hudson and I. Smith. SubArctic UI toolkit user's manual. Tech report, College of Computing, Georgia Tech, 1996.
14. T. Huynh and K. Marriott. Incremental constraint deletion in systems of linear constraints. *Information Processing Letters*, 55:111–115, 1995.
15. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, July 1992.
16. K. Marriott and P. Stuckey. *Introduction to Constraint Logic Programming*. MIT Press, 1997. In preparation.
17. B.A. Myers. The Amulet user interface development environment. In *ACM CHI'96 Conference Companion*, Apr 1996.
18. W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge, 1989.
19. M. Sannella, J. Maloney, B. Freeman-Benson, and A. 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.
20. I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proc. Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.
21. B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM TOPLAS*, 18(1):30–72, Jan 1996.