

# QOCA: A Constraint Solving Toolkit for Interactive Graphical Applications

KIM MARRIOTT

marriott@cs.monash.edu.au

*School of Computer Science & Software Engineering, Monash University, Clayton 3168, Australia*

SITT SEN CHOK

css@cs.monash.edu.au

*School of Computer Science & Software Engineering, Monash University, Clayton 3168, Australia*<sup>1</sup>

**Abstract.** We describe an object-oriented constraint solving toolkit, QOCA, designed for interactive graphical applications. It has a simple yet powerful interface based on the *metric space model* for constraint manipulation. In this model interaction with the constraint solver can occur in three ways: a constraint may be added, a constraint may be deleted, or values for designated “edit” variables may be suggested. Currently, QOCA supports linear arithmetic constraints and two different metrics: the square of the Euclidean distance and Manhattan distance. It provides three solvers, all of which rely on keeping the constraints in solved form and relies on novel algorithms for efficient resolving of constraints during direct manipulation. We provide a thorough evaluation of QOCA, both of the interface design and the speed of constraint solving.

**Keywords:** constraint-based graphics, HCI, linear arithmetic constraint, object oriented

## 1. Introduction

Since the very infancy of computer graphics there has been interest in the use of constraints [18]. This is because constraints allow graphical objects to behave more intelligently since they can be used to model the semantic relationships between diagram components. For instance, in a state transition diagram final states consist of two concentric circles around a block of text. When any one of these components is moved during direct manipulation, the others should follow as well as the end-points of any transitions originating or finishing at the state. Thus the editor will behave as if it “knows” that state transition diagrams are a type of labelled graph. This use of constraints to reflect semantics is not only useful in diagrams, it also holds for many other applications involving 2- or 3-D graphic objects, for instance mathematical equations, tables, architectural plans or documents composed of text and floating figures.

Given the utility of constraints, it is perhaps surprising that they are not more widely used in interactive graphical applications. Apart from CAD systems, most applications provide at best rudimentary constraint solving abilities. We feel that the reasons for this are twofold. First there is the problem of efficiency. Efficient constraint solving techniques, such as those based on local propagation, are suitable for some applications such as simulation but are not powerful enough for many graphical applications since these require inequalities or cyclic constraints. On the other hand, general numerical constraint solving techniques are sufficiently

expressive but naive implementations are too slow, in particular for real time direct manipulation. The second reason is that, perhaps because of concerns about efficiency, many existing constraint solvers are tightly integrated with the graphical application making reuse of the solver in other applications difficult. This means that an application builder who wishes to provide constraints must spend considerable effort and time building their own specialized constraint solver, something most application programmers do not have the time or necessary background in constraint solving to do.

Since 1992, we have been developing an object-oriented constraint solving toolkit, called QOCA, that is expressly designed for interactive graphical applications and which is intended to overcome the above two problems. It is has both C++ and Java versions, each of which contain about 17,000 lines of code.

A major design goal has been to provide an interface that is simple yet flexible enough to be used in a wide variety of applications and which treats constraints as first class objects. This is described in Section 2. The other goal has been to provide constraints that are expressive enough to be useful and yet which can be solved sufficiently quickly for real-time direct manipulation. To some extent there is a synergy between these two goals since, because of the many possible tradeoffs between efficiency and expressiveness, QOCA provides a variety of solvers and a simple common interface allows the application programmer to readily experiment with the different solvers, choosing the appropriate one for their application.

Currently QOCA provides three different solvers. These are described in Section 3. All are based on keeping the current constraints in a tableau and using pivoting to keep them in a solved form. The first, `QCLinEqSolver`, provides linear equalities and uses the square of the (weighted) Euclidean distance to compare solutions. The second, `QCLinIneqSolver`, also uses the square of the Euclidean distance but in addition allows linear inequalities. It is based on linear complementary pivoting [9]. The third, `QCCassSolver`, is based on the Cassowary Algorithm [5]. It also provides linear equalities and inequalities but instead uses the (weighted) Manhattan distance to compare solutions.

We provide a systematic evaluation of QOCA in Section 4. We demonstrate the utility of the interface by describing how it has been employed in five quite different applications: constraint-based graphic editors, error correction in visual language parsing, diagram beautification, graph layout and interactive document layout on the Internet. Our empirical results also demonstrate that the three solvers are sufficiently fast for practical applications.

A preliminary version of this paper appeared in [16]. Apart from this, the most closely related work are our earlier papers on other aspects of QOCA. In [12] we focused on how it could be integrated with an existing graphical editor, Idraw, and in [5] we gave the Cassowary Algorithm. The current paper extends these by detailing new constraint solving algorithms, the metric space model for interaction, the architecture of QOCA and providing the first empirical evaluation of QOCA.

Other related work on constraint solving algorithms for graphical applications are largely formalized in terms of constraint hierarchies [3]. Early solvers provided only local propagation. More recent solvers include UltraViolet [1, 2] which supports

linear equations and certain types of linear inequalities, DETAIL [14, 13] which supports linear equations and the C++ `expr` library which supports linear equations and non-linear equations which are delayed until they become linear [19, 20]. QOCA extends these by supporting arbitrary linear inequalities. In addition we provide an empirical evaluation of our algorithms in the context of a concrete application.

## 2. QOCA's Interface

One of the most striking features of QOCA is its simple interface. To understand the interface design, consider a simple application, that of a constraint-based graphics editor. The editor essentially provides three operations: the user may add objects or constraints, the user may delete an object or constraint, and finally the user may edit or directly manipulate an object. At any point in time, the configuration of the editor consists of a set of graphic objects, a set of constraints over variables in the objects and a current solution which is an assignment to the variable that satisfies the constraints. The variables correspond to graphical attributes of the objects in the diagram and the diagram on the graphics display is, therefore, simply a visual representation of the current solution.

The role of the constraint solver is to find a new solution to the constraints whenever this is required. Reflecting the operations provided by the application, interaction can occur by adding or removing a constraint from the current set of constraints, or the current solution may be manipulated by “suggesting” values for some of the variables. The main difficulty is that usually the constraint system does not have a unique solution, i.e. it is *underconstrained*, but, since we can only display concrete diagrams, the solver must choose a single solution. In the context of graphical applications it is reasonable to expect the constraint solver to find a solution which is “close” to the old solution since during interaction the diagram should not change unnecessarily—it is extremely upsetting for users if, when manipulating one diagram component, other unrelated components also move.

Reflecting this discussion, the QOCA toolkit is based on the *metric space model*, a new metaphor for constraint interaction. In this model, there is a metric which gives the distance between two assignments to the variables, and at any time there is a current set of constraints and a current solution. Mirroring the operations in the graphic editor, interaction in the metric space model can occur in three ways. First, a constraint may be added to the current set of constraints in which case the new solution is the assignment which is closest to the old assignment and which satisfies the new constraint set. Second, a constraint may be deleted from the current set of constraints, in which case the current solution remains unchanged. Finally, the current solution may be manipulated by “suggesting” values for some of the variables. The new solution is the assignment which is closest to the old assignment and to the requested variable values and which satisfies the constraint set. The constraint set remains unchanged.

It is instructive to compare the metric space model to the now classic *constraint hierarchy* approach [3]. A constraint hierarchy consists of a collection of constraints, each of which is labelled with a strength. There is a distinguished strength label

*required:* such constraints must be satisfied. The other strength labels denote preferences. There can be an arbitrary number of such strengths, and constraints with stronger strength labels are satisfied in preference to ones with weaker strength labels.

It is straightforward to formalize the metric space model in terms of a constraint hierarchy. At each point in time the constraint hierarchy consists of the required constraints together with a “weak” constraint for each variable  $v$  in the system of form  $v = a$  where  $a$  is the current value for  $v$ . Less obviously, at least for certain comparators, one can also couch the constraint hierarchy approach in terms of the metric space model. The idea here is to take each non-required equality  $s = t$  and replace it by the required constraint  $s - t = e$  where  $e$  is a distinct new “error” variable whose desired value is always zero. Since inequality constraints may be rewritten to equality constraints by adding a “slack” variable this technique also handles inequalities.

Thus, in a certain formal sense the metric space model and the constraint hierarchy approach are equivalent. The reason that we have based the toolkit on the metric space model rather than the constraint hierarchy model is that we feel it provides a more specialized, higher-level abstraction of the operations required in interactive graphical applications, while the constraint hierarchy model is a more general purpose (but consequently less specific) model for handling underconstrained constraint systems. One advantage of the metric space model is that variables are more structured: they have implicit stay and edit constraints associated with them while these must be explicitly coded in the constraint hierarchy approach. A second advantage is that editing a variable’s value is modelled as a single operation in the metric space model, while the natural way to model variable editing in the constraint hierarchy approach is as a constraint deletion followed by an addition.<sup>2</sup> Apart from simplicity, the advantage of having a distinct operation for editing is that, due to the need to provide continuous feedback during direct manipulation, the performance requirement for editing are much more stringent than for constraint deletion and addition. The metric space model supports better efficiency by allowing the solver to provide a specialized algorithm for editing which can take advantage of the fact that the current solution is “close” to the next solution rather than using algorithms for arbitrary constraint deletion and addition.

QOCA implements the metric space model by means of three main components: constrained variables, called **QCFloats**, linear arithmetic constraints, called **QCConstraints**, and constraint solvers, called **QCSolvers**.

Internally a **QCFloat**,  $v$ , has a *current value*,  $v.val$ , and a *desired value*,  $v.des$ , both of which are floating point numbers. In addition, it has a *stay weight*,  $v.sweight$ , and an *edit weight*,  $v.eweight$ , which, respectively, indicate the importance of leaving the variable unchanged if it is not being edited and the importance of changing it to the desired value when the variable is being edited. Both weights are non-negative floats.

The weights are set by the application programmer when the **QCFloat** is created and cannot be subsequently changed.<sup>3</sup> Applications set the desired value when the variable is created and also when the variable is selected for editing. The solver is

responsible for setting the actual value and will also update the desired value to be the actual value when appropriate. To the application programmer the `QCFloat` appears to have a single value, since they set the desired value using the method `SuggestValue`, call the solver, and then read the actual value using `GetValue`. However, for book keeping reasons, the solver requires `QCFloats` to keep the two values separate.

There are a number of different types of `QCFloats`. The standard `QCFloat` used by the application programmer is the *unrestricted* `QCFloat`. Internally the solvers also make use of *restricted* `QCFloats` whose value must be non-negative and which are used to represent slack variables and artificial variables.

Linear arithmetic constraints, that is `QCConstraints`, can be constructed from `QCFloats`. They have the standard form:

$$a_1 \times x_1 + \dots + a_n \times x_n \text{ rel } c$$

where *rel* is one of `<`, `>`, `>=`, `<=` or `==`, *c* is a float and the *a<sub>i</sub>*s are floats and the *x<sub>i</sub>*s are `QCFloats`. The expression  $a_1 \times x_1 + \dots + a_n \times x_n$  is modelled by the class `QCLinPoly`. An instance of `QCLinPoly` contains a list of `QCLinPolyTerms` where each term  $a_i \times x_i$  is modelled by an instance of the `LinPolyTerm` class. `QCLinPolys` are constructed by adding and subtracting `QCLinPolyTerms`.

The C++ toolkit takes advantage of the C++ facilities for overloading operators. Thus, the standard arithmetic functions, `+`, `*`, and `-` have been overloaded to take `QCFloats` and `QCLinPolyTerms` and return `QCLinPolys` while the relational operators `<`, `>`, `>=`, `<=` and `==` have been overloaded to take a `QCLinPoly` on the LHS and a float on the RHS and return a `QCConstraint`. This allows natural, concise specification of constraints.

Constraint solvers form the heart of QOCA. Interaction with a solver has two modes: *constraint manipulation* and *editing*.

Each solver provides the following Boolean methods for constraint manipulation. Linear constraints can be added to the solver one at a time using `AddConstraint`. With each addition, the solver checks that the new constraint is compatible with the current constraints. If it is not, the constraint is not added and *false* is returned. The solver methods `RemoveConstraint` and `ChangeRHS` respectively allow the application programmer to remove a constraint which is currently in the solver or indicate that its RHS constant has been changed. `ChangeRHS` behaves as if it does a `RemoveConstraint` followed by an `AddConstraint` but the actual implementation may do considerably less work.

QOCA also allows constraint to be added in a block and processed as a whole, rather than one at a time. To do this the programmer calls `BeginAddConstraint`, then calls `AddConstraint` to add each constraint in the block, and finally calls `EndAddConstraint` to signal the end of the block addition. Only at this point is the solver required to check that the constraints added in the block are compatible with those already in the solver. If they are not, then all constraints in the block are removed from the solver and *false* is returned by `EndAddConstraint`. Block addition of constraints is supported because it allows more efficient and more numerically robust satisfaction testing techniques to be used.

After the application programmer has finished adding or removing constraints from the solver they should call the solver method `Solve` to find a new assignment for the variables which is as close as possible to the current solution.<sup>4</sup> The reason for requiring the programmer to perform an explicit call to `Solve` rather than implicitly calling `Solve` after each addition or deletion is that it may be quite expensive, so should only be called when actually needed. The point is that testing for satisfiability does not imply that a solution close to the old solution has been found.

The editing mode is used to modify the values of the “edit” variables. Typically this occurs during direct manipulation. First the application programmer tells the solver which variables are to be edited using multiple calls to `SetEditVar`. Next `BeginEdit` is called. This initializes internal data structures for fast “resolving” of the constraints. Now during manipulation the application programmer repeatedly sets the desired values of the edit variables and then calls the solver function `Resolve` which efficiently computes the new solution to the constraints which is as close as possible to the old solution and to the new desired values of the edit variables. Finally the application programmer calls `EndEdit` to signal the end of the edit phase.

In more detail, if  $v_1, \dots, v_n$  are the `QCFloats`, then `Solve` sets each  $v_i.val$  to the value the variable  $v_i$  takes in a solution to the current constraints which minimizes the objective function

$$\sum_{i=1}^n v_i.weight \times \|v_i - v_i.des\|$$

where the precise metric  $\|\cdot\|$  employed is solver dependent. The objective function employed in `Resolve` is similar except that the weighting for each variable,  $v$ , being edited is  $v.eweight$  rather than  $v.weight$ . One almost always chooses edit weights greater than the stay weights, reflecting the desire to move the variables being edited to the new value at the expense of keeping the value of the other variables unchanged.

Earlier we indicated that QOCA provides different kinds of `QCFloats`. The application programmer can indicate whether they wish the `QCFloat` to be *sedentary* or *nomadic*. Nomadic `QCFloats` have their desired value set to their current value after each call to `EndEdit` or `Solve`. In a sense they forget their previous state, and their future behavior depends only upon their current state. On the other hand, sedentary `QCFloats` remember their desired value, this can only be changed by the application programmer.

Nomadic `QCFloats` are designed to support the metric space model of interaction. For constraint systems containing only nomadic `QCFloats` it is guaranteed that after constraint deletion the current solution will still be a best solution since the objective function measures the distance from the current solution. For the same reason, after editing has finished, the last solution found will still be correct even though the variable is no longer being edited and so has a different associated weight in the metric.

Neither of these statements are true if the constraints contain sedentary `QCFloats`. In this case, `Solve` may need to be called after constraints have been removed since

there may now exist a solution closer to the sedentary variables' desired values. Furthermore, anomalous behaviour can result when a sedentary `QCFloat` is edited if its stay and edit weight are different. This is because after `EndEdit` is called the current solution may not be the optimal solution with respect to the stay weights although it is optimal with respect to the edit weights. Thus, sedentary `QCFloats` have only a single weight used as both the stay and edit weight for that variable.

Sedentary `QCFloats` are principally provided to support non-required constraints in the constraint hierarchy model. If the desired constraint  $s = t$  is modelled by  $s - t = e$  where the desired value of the error variable is 0, it is important that  $e$  maintains 0 as its desired value. Thus  $e$  must be declared to be a sedentary `QCFloat`.

Each solver also provides the methods `AddVar` and `RemoveVar`. These respectively add and remove variables from the solver. Variables are implicitly added to the solver the first time a constraint with those variables is added to the solver, but an explicit call to `AddVar` allows variables which are currently not involved in any constraints to be added to a solver. `RemoveVar` should only be called if there are no constraints in the solver referring to that variable.

The final method a solver must provide is `Recompute`. This is used to recompute internal data structures and resolve for the current constraints in a numerically stable fashion. The problem is that if constraints are solved incrementally, it may not be possible to (efficiently) do this in a numerically stable fashion, so rounding errors build up. `Recompute` will resolve the current constraint system in a non-incremental fashion, hopefully reducing the build-up of errors.

EXAMPLE: Consider a (rather simple) diagram consisting of a point  $(xm, ym)$  and a line from  $(xl, yl)$  to  $(xu, yu)$  in which the point is constrained to lie at the midpoint of the line. The following C++ program fragment creates the variables and constraints, adds them to the solver and calls `Solve` to compute the initial position. The constructor for `QCFloats` takes three arguments: the stay and edit weights together with an initial desired value. Note that both  $xm$  and  $ym$  have a stay weight of zero indicating that they are “dependent variables,” although they have a non-zero edit weight (otherwise, suggesting new values for them during editing would never have any affect). By default the `QCFloat` constructed is nomadic and unrestricted.

Next the program chooses  $xm$  and  $ym$  to be the edit variables, and then repeatedly samples the mouse to find the desired values and calls `Resolve` to compute the new value until the user releases the mouse button, which finishes the edit cycle.

```
QCFloat xl(1,1000,45), xm(0,1000,0), xu(1,1000,60),
        yl(1,1000,45), ym(0,1000,0), yu(1,1000,60);
QCConstraint xcon = (1*xl + 1*xu - 2*xm == 0),
              ycon = (1*yl + 1*yu - 2*ym == 0);
QCLinEqSolver solv;

solv.AddConstraint(xcon);
solv.AddConstraint(ycon);
```

```

solv.Solve();
DrawLine(x1.GetValue(),y1.GetValue(),xu.GetValue(),yu.GetValue());

solv.SetEditVar(xm);
solv.SetEditVar(ym);
solv.BeginEdit();
while (mouse.button.down) {
    xm.SuggestValue(mouse.x);
    ym.SuggestValue(mouse.y);
    solv.Resolve();
    DrawLine(x1.GetValue(),y1.GetValue(),xu.GetValue(),yu.GetValue());
}
solv.EndEdit();

```

□

### 3. The Constraint Solvers

The QOCA toolkit is designed to provide a variety of solvers. These may provide different types of constraints, use different metrics or employ different algorithms and time/space tradeoffs in constraint solving. Currently QOCA provides three true solvers and two auxiliary solvers. We now briefly describe these.

#### 3.1. Solved Form and Tableaus

All of the solvers are based on transforming the original linear arithmetic constraints into a *solved form*. Each constraint  $c_i$  in the solved form is required to be either of the form  $0 = 0$  or of the form  $x_i + \sum_{j=1}^m a_{ij}y_j = b_i$  where the variable  $x_i$  occurs only once in  $c_i$  and the  $y_1, \dots, y_m$  are variables and the  $a_{ij}$  and  $b_i$  are constants. If the variable  $x_i$  is restricted, i.e. a slack variable, the RHS constant  $b_i$  is required to be non-negative. The variables  $x_1, \dots, x_n$  are said to be *basic* and the  $y_1, \dots, y_m$  are said to be *parameters*. A constraint of the form  $0 = 0$  indicates that a *redundant* constraint is present, i.e. a constraint implied by the other constraints in the system.

To support manipulation of constraints into their solved form QOCA provides the `QCTableau` class. Conceptually, the standard tableau contains three parts: the original constraints, the solved form, and the *quasi-inverse*. The quasi-inverse is a matrix representing the elementary row operations which have been performed on the original constraints to obtain the normal form.

EXAMPLE: Consider computing the solved form of the equations

$$\begin{aligned}
 x &= 1, \\
 y - x - z &= 0, \\
 2x - 2 &= 0, \\
 z &= 0.
 \end{aligned}$$

The solver will perform row operations to transform this into solved form, and at the same time the tableau will also perform the row operations on the *quasi-inverse* matrix which is initially the identity matrix. That is we start with the matrix (where variable names have been included for clarity).

$$\left[ \begin{array}{cccc|ccc} & & & & x & y & z & RHS \\ \hline 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & -1 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 & 2 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{array} \right]$$

We can transform this to solved form by using the top equation to eliminate  $x$ , the second equation to eliminate  $y$ , and the fourth equation to eliminate  $z$  giving:

$$\left[ \begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ -2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{array} \right]$$

Note that the third equation is redundant. □

The point to note is that the *quasi-inverse* details how the original constraints have been used to compute the solved form. To determine how the  $j$ th equation in the solved form has been computed we examine the  $j$ th row of the quasi-inverse. For example, in this case looking at the second row, we know that the second constraint in the solved form is the sum of the 1st, 2nd and 4th original equations.

The primary reason for computing the quasi-inverse is that it allows incremental deletion of constraints. Imagine we wish to remove the original  $j$ th equation. The  $j$ th column of the quasi-inverse details how the  $j$ th equation has been used to compute the solved form. We perform a pivot operation on the  $j$ th column of the quasi-inverse, say at row  $i$ . This eliminates the  $j$ th constraint from all except the  $i$ th constraint in the solved form. We now remove the  $i$ th row and  $j$ th column from the quasi-inverse and the  $i$ th constraint from the solved form, and we have a new solved form which does not make use of the original  $j$ th constraint. When choosing the row to pivot on we prefer to use a row corresponding to a redundant constraint.

EXAMPLE: For instance, we wish to delete the first original constraint. We look at the non-zero entries in the 1st column. Row 3 corresponds to the redundant  $0 = 0$  equation in the solved form so we choose row 3 to pivot on. This gives

$$\left[ \begin{array}{cccc|ccc} 0 & 0 & 1/2 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1/2 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & -1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{array} \right]$$

Eliminating column 1 from the quasi-inverse and row 3 from the quasi-inverse and solved form gives the new solved form and quasi-inverse. Note that the solved form has not changed. □

The `QCTableau` class is at the heart of the solvers. Its methods are naturally grouped into three kinds. The first kind of methods are for accessing or modifying the original constraints and variables. They can be used to add, delete or change the RHS of an original constraint. These changes are reflected in the solved form, which may no longer be “solved.” The second kind of methods are for accessing the solved form. They can be used to iterate through the variables or constraints in the solved form, determine the basic variable for a particular constraint, indicate if a constraint is currently in solved form, and to find the RHS or coefficient of a variable in a constraint in the solved form. The final kind of methods are for performing elementary row operations on the solved form. They can be used to “pivot” on a variable in a constraint, i.e. making that variable basic in that constraint and eliminating it from all other constraints, and eliminating the basic variables from a constraint which has just been added to the tableau.

Careful implementation of the tableau class is vital for reasonable performance. Indeed, because of the multiple uses of the tableau class, the QOCA toolkit provides a number of variants. One of these is the *non-incremental* tableau which does not allow incremental deletion of constraints from the tableau since it does not contain the quasi-inverse.

One of the most important implementation decisions for the standard (incremental) tableau is whether to explicitly compute the solved form or rather to implicitly compute it as needed by multiplying the original constraints by the quasi-inverse. Note that this is the idea behind the revised simplex method. We have experimented with three different implementations. The first explicitly computes the solved form, the second implicitly computes the solved form, while the third implicitly computes the variable coefficients but explicitly compute the RHS constants in the solved form. In all implementations of the tableau a matrix is used to represent the quasi-inverse, the original constraints and the solved form. In practice, we have found that the first design is more time efficient (but clearly less space efficient). Another design decision is whether to use a dense or to use a sparse matrix representation for the original constraints, solved form and the quasi-inverse. Again we have experimented with different options, and have found a sparse representation to be best for representing the solved form and the original constraints and, in most applications, even the quasi-inverse.

### 3.2. The Simple Linear Equality Solver: `QCLinEqSystem`

The most basic solver is `QCLinEqSystem`. This provides incremental addition and deletion of linear equations. Although it provides `Solve` and `Resolve`, the suffix “`System`” indicates that it does not directly support the metric system model. Instead the application programmer should use the solver `QCLinEqSolver`. This, however, is implemented using `QCLinEqSystem` so we shall first explain how `QCLinEqSystem` works.

`QCLinEqSystem` is built around a single tableau. The key invariant in `QCLinEqSystem` is that after each operation the constraints are left in solved form. We now briefly describe the solver’s implementation. The methods `SetEditVar`, `AddVar`

and `RemoveVar` have the obvious implementation. The other methods are implemented as follows:

**AddConstraint:** Incremental addition of a linear equation is performed by using an incremental version of Gauss-Jordan elimination (see for instance [17]).

**RemoveConstraint:** Constraint removal is handled by using the quasi-inverse as detailed above.

**ChangeRHS:** The original constraint in the tableau is modified and by employing the quasi-inverse the differential change to the RHS values of the solved form is efficiently computed. Constraints in the solved form that were redundant are checked to ensure they are still redundant and have not become unsatisfiable.

**BeginAddConstraint** and **EndAddConstraint:** Batch mode addition of constraints is currently performed by multiple calls to `AddConstraint`.

**BeginEdit:** First this performs pivots to make edit variables parameters, if this is possible, and then computes *DepVars*, the set of basic variables which depend on the edit variables.

**Solve** and **Resolve:** These do not directly support the metric space model. Instead the solution computed depends on the current solved form. The solver simply uses the value of the parametric variables to compute values for the basic variables. The effect of this is to ignore the desired values for basic variables. One subtlety is that **Resolve** computes the new values of the basic variables in *DepVars* differentially. We note that in theory this may lead to a buildup of rounding errors, but this does not seem to occur in practice.

**EndEdit:** This sets the desired value of all nomadic variables to be their actual value and clears the set of edit variable.

**Recompute:** This implements a numerically stable version of Gauss-Jordan elimination.

### 3.3. Linear Equality Solver: `QCLinEqSolver`

The solver `QCLinEqSolver` is implemented in terms of `QCLinEqSystem`. It provides incremental addition and deletion of linear equalities as well as a `Solve` and `Resolve` which support the metric system model. The square of the Euclidean distance from each variable's desired value is used as the metric.

Constraints are kept inside a private `QCLinEqSystem`. Thus, the methods `Recompute`, `AddConstraint`, `RemoveConstraint`, `ChangeRHS`, `BeginAddConstraint`, and `EndAddConstraint` are directly provided by the underlying `QCLinEqSystem`.

**Solve:** Intuitively this is quite simple, we must find the solution which minimizes the function,

$$f(\mathbf{v}) = \sum_{i=1}^n v_i.weight \times (v_i - v_i.des)^2,$$

with respect to the constraints  $C$  in the solver where  $\mathbf{v}$  is the vector of `QCFloats`,  $v_1, \dots, v_n$ . We do this by using the solved form of  $C$  to eliminate the basic variables from  $f$ . This gives a quadratic polynomial, say  $f_e(\mathbf{y})$ , over the parametric variables  $y_1, \dots, y_m$ . The minimal value for  $f_e$  will occur at any point where all partial

derivatives are zero. Thus to find the value for the parameters which minimizes the original system we set up another `QCLinEqSystem` with a constraint  $\partial f_e / \partial y_i = 0$  for each parameter  $y_i$ . For efficiency a non-incremental tableau is used for solving these equations. Once the value for each  $y_i$  is found, these are passed as desired values into the original `QCLinEqSystem` to find the total solution.

More exactly, if the solved form is

$$\mathbf{x} + \mathbf{A}\mathbf{y} = \mathbf{b}$$

and the vector of desired values for  $\mathbf{x}$  is  $\mathbf{d}^x$  and for  $\mathbf{y}$  is  $\mathbf{d}^y$  with stay weights  $\mathbf{w}^x$  and  $\mathbf{w}^y$  respectively, then the derivative  $\nabla f_e(\mathbf{y})$  is

$$2(\mathbf{A}^T \begin{bmatrix} w_1^x & & 0 \\ & \ddots & \\ 0 & & w_k^x \end{bmatrix} (\mathbf{d}^x - \mathbf{b} - \mathbf{A}\mathbf{y}) + \begin{bmatrix} w_1^y & & 0 \\ & \ddots & \\ 0 & & w_m^y \end{bmatrix} (\mathbf{y} - \mathbf{d}^y)).$$

**Resolve:** This is quite similar to **Solve**. One difference is that edit weights are used instead of stay weights for edit variables. A more important difference is that in **BeginEdit** those parameters that depend upon the edit variables are identified. When constructing the partial derivatives we need only consider these parameters, since the other parameters will maintain their current value and so can be ignored in **Resolve**. One difficulty is that the constraint system  $\nabla f_e(\mathbf{y}) = \mathbf{0}$  depends on the desired values for the variables. In particular, the system depends upon the desired values for the edit variables and these can change each time **Resolve** is called. Clearly we do not want to create and solve a new system  $\nabla f_e(\mathbf{y}) = \mathbf{0}$  for each call to **Resolve**. Instead in **BeginEdit** we compute the parametric derivative  $\nabla f_e(\mathbf{y}, \mathbf{d})$  where  $\mathbf{d}$  is a vector of variables, one for each edit variable's desired value. We are careful to ensure that when pivoting the  $\mathbf{d}$ s remain parameters. Then in **Resolve** we simply solve for the parameters  $\mathbf{y}$  given the new values for the  $\mathbf{d}$ s (which is simply the new desired value of the corresponding edit variable). In turn we resolve for the values of the  $\mathbf{x}$  given values for the  $\mathbf{y}$  in the original tableau. Thus **Resolve** in `QCLinEqSolver` actually calls **Resolve** in each of its internal `QCLinEqSystems`.

**EXAMPLE:** Imagine that we have the constraints from Example 2 and that the solved form is

$$xl - 2xm + xu = 0, \quad ym - \frac{1}{2}yl - \frac{1}{2}yu = 0$$

where  $xl$  and  $ym$  are basic. The objective function  $f$  is

$$(xl - 45)^2 + (xu - 60)^2 + (yl - 45)^2 + (yu - 60)^2 + 1000(xm - dx)^2 + 1000(ym - dy)^2$$

where  $xm$  and  $ym$  are edit variables and  $dx$  and  $dy$  are the new variables representing their desired values. Eliminating the basic variables  $xl$  and  $ym$  gives:

$$(2xm - xu - 45)^2 + (xu - 60)^2 + (yl - 45)^2 + (yu - 60)^2 + 1000(xm - dx)^2 + 1000(\frac{1}{2}yl + \frac{1}{2}yu - dy)^2$$

Differentiating with respect to each of the parameters,  $xm$ ,  $xu$ ,  $yl$  and  $yu$  (but not with respect to  $dx$  and  $dy$ ) gives the system

$$\begin{aligned} 4(2xm - xu - 45) + 2000(xm - dx) &= 0, \\ -2(2xm - xu - 45) + 2(xu - 60) &= 0, \\ 2(yl - 45) + 1000\left(\frac{1}{2}yl + \frac{1}{2}yu - dy\right) &= 0, \\ 2(yu - 60) + 1000\left(\frac{1}{2}yl + \frac{1}{2}yu - dy\right) &= 0. \end{aligned}$$

□

The remaining methods—`EndEdit`, `AddVar`, `SetEditVar` and `RemoveVar`—have the natural definitions.

It is instructive to consider the size of the tableaus being manipulated in each of the `QCLinEqSystems` in terms of the number of linear equations  $E$  and variables  $V$  which the user has added to the system. The core `QCLinEqSystem` stores exactly these constraints so it will also have  $E$  constraints and  $V$  variables. The `QCLinEqSystem` for the derivatives constructed in `Solve` will have a constraint for each parameter of the original system. Thus, assuming there are no redundant equations, this `QCLinEqSystem` will have  $V - E$  constraints in  $V - E$  variables.

### 3.4. The Simplex Solver: `QCLinIneqSystem`

The most basic solver provided by QOCA which supports linear inequalities as well as linear equalities is `QCLinIneqSystem`. Like `QCLinEqSystem`, this solver is not intended for direct use by the application programmer since it does not support the metric space model, rather it is used to implement the two solvers for inequalities, `QCLinIneqSolver` and `QCCassSolver`, that do support the metric space model. `QCLinIneqSystem` contains a single tableau to which constraints are added. The interesting methods are described below.

**AddConstraint:** Incremental addition of a linear equation or inequality is performed by using an incremental version of Gauss-Jordan elimination and phase I of the simplex algorithm (see for instance [17, 5]). When a constraint is added, it is first rewritten to an equation by adding a slack variable if the constraint is an inequality. Note that slack variables are restricted `QCFloats` since they must be non-negative. Next basic variables in the current solved form are eliminated. If the resulting equation contains only slack variables, then an incremental version of phase I of the simplex algorithm is used to determine satisfiability (which takes advantage of the fact that, since only the newly added constraint can have a negative RHS, at most one artificial variable is required). Otherwise, Gauss-Jordan elimination is used to make one of the remaining non-slack variables basic.

It follows from the algorithm that the solved form will always consist of two components: those constraints in which the basic variables are not slacks and whose parameters can either be restricted or unrestricted, and those whose basic variables are slacks and whose parameters are all restricted variables. Thus the solved form has the form

$$\mathbf{x}^{\text{ur}} + \mathbf{A}^1 \mathbf{y}^{\text{ur}} + \mathbf{A}^2 \mathbf{y}^{\text{sl}} = \mathbf{b}^{\text{ur}} \quad \wedge \quad \mathbf{x}^{\text{sl}} + \mathbf{A}^3 \mathbf{y}^{\text{sl}} = \mathbf{b}^{\text{sl}}$$

where  $\mathbf{x}^{\text{ur}}$  is the vector of “unrestricted” basic variables,  $\mathbf{y}^{\text{ur}}$  is the vector of “unrestricted” parameters,  $\mathbf{x}^{\text{sl}}$  is the vector of basic variables which are slacks,  $\mathbf{y}^{\text{sl}}$  is the vector of parameters which are slacks and the RHS constants  $\mathbf{b}^{\text{sl}}$  are required to be non-negative.

**RemoveConstraint:** The algorithm for constraint removal is essentially that given in [15]. It is similar to that used in `QCLinEqSystem`. The only extra complication is the need to ensure that pivoting to remove the constraint maintains the invariant that the RHS constants associated with slack basic variables are non-negative and that for any constraint whose basic variable is slack, all variables in the constraint are slack.

**ChangeRHS:** Similar to `QCLinEqSystem`. The new RHS values of the solved form are computed. If any of the RHS constants associated with slack basic variables have become non-negative, phase I of the simplex is called to check for satisfiability.

**BeginAddConstraint** and **EndAddConstraint:** Addition of a constraint in batch mode is performed as follows. First, basic variables in the constraint are eliminated. If it contains only slack variables, then it is simply tagged as new and placed in the tableau. If it contains a non-slack variable, incremental Gauss-Jordan elimination is performed. After all constraints in the batch have been added, a non-incremental version of phase I of the simplex is used to test satisfiability of the constraints in the tableau which were tagged as new.

**Solve:** This does not directly support the metric space model. Instead it uses phase II of the simplex algorithm to minimize

$$\sum_{i=1}^n v_i.\text{weight} \times (v_i - v_i.\text{des})$$

with respect to the linear constraints  $C$  in the solver with `QCFloats`  $v_1, \dots, v_n$ .

Note that the values found by `Solve` for  $v_1, \dots, v_n$  do not depend on the desired values for the variables since minimizing the above expression is equivalent to minimizing

$$\sum_{i=1}^n v_i.\text{weight} \times v_i.$$

Thus this solver does not need to support `Resolve`, `BeginEdit`, `EndEdit`, and `SetEditVar` since changing the desired value of a `QCFloat` can never lead to a change in its value!

**Recompute:** This implements a numerically stable version of Gauss-Jordan elimination as well as a non-incremental version of phase I of the simplex.

### 3.5. Linear Complementary Pivoting: `QCLinIneqSolver`

QOCA provides two solvers which support linear inequalities as well as equalities and which provide a `Solve` and `Resolve` which supports the metric system model. They differ in the choice of metric used to find the closest solution and the underlying algorithms. The first solver we shall look at, `QCLinIneqSolver`, like `QCLinEqSolver` uses the square of the Euclidean distance as the metric.

In this solver, constraints are kept inside a private `QCLinIneqSystem`. This means `AddConstraint`, `RemoveConstraint`, `ChangeRHS`, `BeginAddConstraint`, `EndAddConstraint`, and `Recompute` are directly provided by the underlying `QCLinIneqSystem`. `Solve`: This must find the solution that minimizes the function,

$$f(\mathbf{v}) = \sum_{i=1}^n v_i.weight \times (v_i - v_i.des)^2$$

with respect to the linear arithmetic constraints in the solver where  $\mathbf{v}$  is the vector of `QCFloats`,  $v_1, \dots, v_n$ . This is an example of a convex quadratic problem. Arguably the simplest approach to solving such 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 optimum. The earliest approach is probably the Dantzig-Wolfe algorithm, although here we shall use a modification of Lemke's algorithm [9].

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. Since the derivative of a quadratic optimization function is linear, both the dual and the primal problem consist of linear arithmetic constraints and so a variant of the simplex algorithm can be used to solve their conjunction.

We first consider the standard quadratic programming problem in which all constraints are inequalities and all variables are restricted to take non-negative values. We can write the primal problem as:

$$PP: \quad \mathbf{A}\mathbf{y} + \mathbf{x} = \mathbf{b}$$

where  $\mathbf{y}$  are the original variables and  $\mathbf{x}$  are the slack variables. Let  $f$  be the expression to be minimized. Note that it does not contain occurrences of the slack variables. Then the dual problem is

$$DP: \quad \mathbf{t} - \mathbf{A}^T \mathbf{z} = \nabla f(\mathbf{y})$$

where  $\mathbf{z} \geq \mathbf{0}$  are the dual variables (one for each equation in the primal problem) and  $\mathbf{t} \geq \mathbf{0}$  are the dual slack variables. The combined problem *CP* is the conjunction of the dual and primal problem plus the constraints that  $\mathbf{x} \cdot \mathbf{z} = \mathbf{0}$  and  $\mathbf{y} \cdot \mathbf{t} = \mathbf{0}$ . Note that the last constraints mean that in the combined problem every variable has a *complementary* variable and in a solution to the problem a variable and its complementary variable cannot both be positive.

EXAMPLE: Imagine that we have three variables,  $x_l$ ,  $x_r$  and  $x_m$ , which are restricted to take non-negative values,  $x_m$  is in the middle of  $x_l$  and  $x_r$ ,  $x_l$  is to the

left of  $x_r$  and  $x_r$  is less than 100. After rewriting into solved form, our primal problem is:

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

Now we can choose to treat each of these equations as an inequality which has the basic variable as its slack. The only thing to remember is that we must eliminate these basic variables from the objective function. Assume that  $x_l.des = 40$ ,  $x_m.des = 60$  and  $x_r.des = 60$ . After elimination we obtain the function

$$f_e = (x_l - 40)^2 + \left(\frac{1}{2}x_l + \frac{1}{2}x_r - 60\right)^2 + (x_r - 60)^2.$$

Putting the primal and dual together we obtain:

$$\begin{array}{rcl} x_m & -\frac{1}{2}x_l & -\frac{1}{2}x_r & & & = & 0, \\ s_1 & +x_l & -x_r & & & = & 0, \\ s_2 & & +x_r & & & = & 100, \\ t_1 & -\frac{5}{2}x_l & -\frac{1}{2}x_r & +\frac{1}{2}z_1 & -z_2 & = & -150, \\ t_2 & -\frac{1}{2}x_l & -\frac{5}{2}x_r & +\frac{1}{2}z_1 & +z_2 & -z_3 & = & -180. \end{array}$$

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

$$x_l \leftrightarrow t_1 \quad x_m \leftrightarrow z_1 \quad x_r \leftrightarrow t_2 \quad s_1 \leftrightarrow z_2 \quad s_2 \leftrightarrow z_3.$$

Unfortunately the complete problem is not in solved form since some of the RHS constants are negative. To transform it into a solved form we subtract an artificial variable  $v$  from every equation and then pivot on the row with the largest negative constant, making  $v$  basic in that row. We obtain:

$$\begin{array}{rcl} x_m & -t_2 & & +2x_r & -\frac{1}{2}z_1 & -z_2 & +z_3 & = & 180, \\ s_1 & -t_2 & +\frac{3}{2}x_l & +\frac{3}{2}x_r & -\frac{1}{2}z_1 & -z_2 & +z_3 & = & 180, \\ s_2 & -t_2 & +\frac{1}{2}x_l & +\frac{7}{2}x_r & -\frac{1}{2}z_1 & -z_2 & +z_3 & = & 280, \\ t_1 & -t_2 & -2x_l & +x_r & & -2z_2 & +z_3 & = & 30, \\ v & -t_2 & +\frac{1}{2}x_l & +\frac{5}{2}x_r & -\frac{1}{2}z_1 & -z_2 & +z_3 & = & 180. \end{array}$$

We have now obtained a solved form which is 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  $x_r$  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.  $\square$

Unfortunately our case is more complex because we also have unrestricted variables and so is not in the standard form found in the literature on convex quadratic programming. The dual problem is actually the first-order conditions necessary for optimality, often called the Karush-Kuhn-Tucker (KKT) conditions. In general, if we wish to minimize  $f(\mathbf{x})$  with respect to the system of constraints

$$\begin{aligned} h_i(\mathbf{x}) &= 0, \quad i = 1, \dots, m \\ g_j(\mathbf{x}) &\geq 0, \quad j = 1, \dots, t \end{aligned}$$

the Karush-Kuhn-Tucker conditions are

$$\begin{aligned} \nabla f(\mathbf{x}) - \sum_{i=1}^m \mu_i \nabla h_i(\mathbf{x}) - \sum_{j=1}^t \pi_j \nabla g_j(\mathbf{x}) &= \mathbf{0}, \\ \pi_j &\geq 0, & j = 1, \dots, t \\ \pi_j \cdot g_j &\geq 0, & j = 1, \dots, t \end{aligned}$$

where  $\mu_i$  is the dual variable for  $h_i(\mathbf{x})$  and  $\pi_j$  the dual variable for  $g_j(\mathbf{x})$ .

Now the solved form for our problem will have the special form identified earlier:

$$\mathbf{x}^{\text{ur}} + \mathbf{A}^1 \mathbf{y}^{\text{ur}} + \mathbf{A}^2 \mathbf{y}^{\text{sl}} = \mathbf{b}^{\text{ur}} \quad \wedge \quad \mathbf{x}^{\text{sl}} + \mathbf{A}^3 \mathbf{y}^{\text{sl}} = \mathbf{b}^{\text{sl}}$$

Treating the  $\mathbf{x}^{\text{sl}}$  as slack variables we therefore have the primal problem

$$\mathbf{x}^{\text{ur}} + \mathbf{A}^1 \mathbf{y}^{\text{ur}} + \mathbf{A}^2 \mathbf{y}^{\text{sl}} = \mathbf{b}^{\text{ur}} \quad \wedge \quad (1)$$

$$-\mathbf{A}^3 \mathbf{y}^{\text{sl}} + \mathbf{b}^{\text{sl}} \geq \mathbf{0} \quad \wedge \quad (2)$$

$$\mathbf{y}^{\text{sl}} \geq \mathbf{0} \quad (3)$$

We assume that the basic variables  $\mathbf{x}^{\text{ur}}$  have been eliminated from the function  $f_e$  to be minimised. Thus  $f_e$  is a function of  $\mathbf{y}^{\text{ur}}$  and  $\mathbf{y}^{\text{sl}}$ .

We let the dual variables be  $\mu$ , one for each equation in Equation 1,  $\mathbf{z}$ , one for each inequality in Equation 2, and  $\mathbf{t}$ , one for each slack in Equation 3. In the dual problem we have for the  $\mathbf{x}^{\text{ur}}$ ,

$$\nabla f_e(\mathbf{x}^{\text{ur}}) - \mu = \mathbf{0}.$$

Since the  $\mathbf{x}^{\text{ur}}$  do not occur in  $f_e$ ,  $\nabla f_e(\mathbf{x}^{\text{ur}}) = \mathbf{0}$ , and so  $\mu = \mathbf{0}$ . For the  $\mathbf{y}^{\text{ur}}$  we have that

$$\nabla f_e(\mathbf{y}^{\text{ur}}) - \mathbf{A}^{1T} \mu = \mathbf{0}.$$

Since  $\mu = \mathbf{0}$ , this is equivalent to  $\nabla f_e(\mathbf{y}^{\text{ur}}) = \mathbf{0}$ . For the  $\mathbf{y}^{\text{sl}}$  we have

$$\nabla f_e(\mathbf{y}^{\text{sl}}) - \mathbf{A}^{2T} \mu + \mathbf{A}^{3T} \mathbf{z} - \mathbf{t} = \mathbf{0},$$

which reduces to

$$\nabla f_e(\mathbf{y}^{\text{sl}}) + \mathbf{A}^{3T} \mathbf{z} - \mathbf{t} = \mathbf{0}.$$

Therefore, the dual problem is equivalent to:

$$\nabla f_e(\mathbf{y}^{\text{ur}}) = \mathbf{0} \wedge \nabla f_e(\mathbf{y}^{\text{sl}}) = \mathbf{t} - \mathbf{A}^{\mathbf{3}T} \mathbf{z}$$

together with the restriction and complementary conditions

$$\begin{aligned} \mathbf{z} &\geq 0, \\ \mathbf{t} &\geq 0, \\ \mathbf{t} \cdot \mathbf{y}^{\text{sl}} &= 0, \\ \mathbf{z} \cdot \mathbf{x}^{\text{sl}} &= 0. \end{aligned}$$

We solve this by first using the complementary pivot algorithm to find values for  $\mathbf{y}^{\text{sl}}$ ,  $\mathbf{y}^{\text{ur}}$  and  $\mathbf{x}^{\text{sl}}$  by solving

$$\nabla f_e(\mathbf{y}^{\text{ur}}) = \mathbf{0} \wedge \mathbf{x}^{\text{sl}} + \mathbf{A}^{\mathbf{3}} \mathbf{y}^{\text{sl}} = \mathbf{b}^{\text{sl}} \wedge \nabla f_e(\mathbf{y}^{\text{sl}}) = \mathbf{t} - \mathbf{A}^{\mathbf{3}T} \mathbf{z}$$

and then performing back substitution of the parameter values into the original `QCLinIneqSystem` in order to find values for  $\mathbf{x}^{\text{ur}}$ . Note that a non-incremental tableau is used when solving the dual problem.

**Resolve:** It is straightforward to modify the algorithm for `Solve` so that it is incremental for resolving. Changing the desired variable values only changes the RHS 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. In order to be able to update the RHS efficiently, as in `QCLinEqSolver`, we introduce a new variable for each edit variable which acts as a placeholder for the desired value.

Again it is interesting to think about the number of constraints and variables inside the internal `QCLinIneqSystem`. If the programmer adds  $E$  equalities and  $I$  inequalities in variables  $V$ , then the internal tableau of the `QCLinIneqSystem` will have  $E + I$  constraints in  $I + V$  variables. Now consider the additional tableau constructed when `Solve` is called. We let the number of unrestricted basic variables be  $X^{\text{ur}}$ , the number of unrestricted parameters be  $Y^{\text{ur}}$ , the number of slack basic variables be  $X^{\text{sl}}$ , and the number of slack parameters be  $Y^{\text{sl}}$ . The supplementary tableau has  $X^{\text{sl}} + Y^{\text{sl}} + Y^{\text{ur}} = I + Y^{\text{ur}}$  constraints in  $Y^{\text{ur}} + 2I$  variables.

### 3.6. Cassowary Solver: `QCCassSolver`

The final solver provided in the QOCA toolkit is `QCCassSolver`, based on the Cassowary Algorithm described in [5]. Like `QCLinIneqSolver`, this provides linear inequality and equality constraints and its implementation is based on `QCLinIneqSystem`. Unlike `QCLinIneqSolver`, `QCCassSolver` measures the distance between two solutions using the Manhattan, i.e. rectilinear, distance rather than the square of the Euclidean distance.

The reasons for providing two solvers for linear inequality and equality constraints are twofold. First, different metrics are suitable for different applications. Second,

the Cassowary Algorithm is quite different to the complementary pivot algorithm, so it is instructive to compare their relative speed.

The key idea behind the Cassowary Algorithm is to minimize the objective function

$$\sum_{i=1}^n v_i.weight \times |v_i - v_i.des|$$

by solving a related linear programming problem. This problem is obtained by first adding to the original problem new restricted variables  $\delta_i^+$  and  $\delta_i^-$  for each of the original `QCFloats`  $v_i$ . These are the positive and negative error respectively of the variable  $v_i$ . Then for each variable  $v_i$  we associate an *error equality*

$$v_i + \delta_i^+ - \delta_i^- = v_i.des.$$

The solution to the original problem is found by minimizing the linear objective function

$$\sum_{i=1}^n v_i.weight \times (\delta_i^+ + \delta_i^-)$$

with respect to the original constraints together with the error equalities. We now briefly describe the interesting methods:<sup>5</sup>

**AddConstraint:** This adds the constraint together with an error equality for each new variable to an internal `QCLinIneqSystem`.

**RemoveConstraint:** This removes the constraint as well as the associated error equalities from the internal `QCLinIneqSystem`.

**Solve:** Uses phase II of the simplex algorithm to optimize with respect to the objective function given above using a call to the `Solve` method of the internal `QCLinIneqSystem`.

**BeginEdit:** Modifies the objective function so that the edit weight is used to multiply the error variables associated with edit variables rather than the stay weight.

**Resolve:** Changing the desired value for an edit variable means that the RHS constant of the error equation must be changed. This is achieved by calling the `ChangeRHS` method of the internal `QCLinIneqSystem`. If this causes the RHS of any restricted variable in the solved form to become negative, the dual simplex algorithm is used to restore feasibility.

**EndEdit:** This sets the desired value of each variable to its actual value. In order to reflect this change, the `ChangeRHS` method of the internal `QCLinIneqSystem` is used to appropriately modify the RHS constants of the error equations.

We note that the algorithms for `Resolve` and `EndEdit` are somewhat simpler than those given in [5]. This is because we can simply change the RHS constants of the error equations and then use the `ChangeRHS` method of the internal `QCLinIneqSystem` to appropriately update the RHS constants in the solved form. This is easy because the tableau contains the quasi-inverse. In contrast the algorithm in [5] must explicitly modify the solved form.

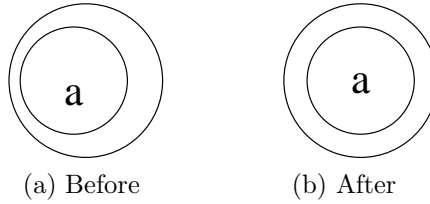


Figure 1. Error Correction When Recognising a Final State

It is useful to think about the number of constraints and variables inside the internal `QCLinIneqSystem`. If the programmer adds  $E$  equalities and  $I$  inequalities in  $V$  variables to `QCCassSolver`, then the internal tableau of the `QCLinIneqSystem` will have  $E + I + V$  constraints in  $I + 3V$  variables.

#### 4. Evaluation

There are two aspects of the toolkit which warrant evaluation. The first is the design of the interface, the second is performance of the constraint solvers. We look at these in turn.

Clearly the interface is reasonably simple. The question is whether it is sufficiently flexible to be used in a variety of applications. The original motivation for the interface design and the associated metric space model came from the integration with the graphic editor `Idraw` [12]. Since then we have used `QOCA` in a number of other graphical applications which we now detail.

`QOCA` has been employed for three different purposes in the Penguins system [6, 7, 8]. Given a grammatical specification of a visual language, such as state transition diagrams or mathematical equations, the Penguins system automatically generates an incremental parser for the visual language. This is integrated into a generic editing tool resulting in a customised graphics editor for that particular visual language.

One use of `QOCA` is for error correction during parsing. Error correction is required since diagram components may be placed at any location and so a diagram almost invariably contain errors because, for instance, humans cannot draw perfectly concentric circles or coincident arrows. The parsing process employed in Penguins “relaxes” the constraints during recognition but then `QOCA` is used to find the closest placement of the graphical objects to their original placement which satisfies the constraints.

As an example, consider what happens if a user of the Penguins generated customised editor for state transition diagrams creates the circles and text shown in Figure 1(a). `QCFloats` are created for the geometric attributes of each graphical object, for example the  $x$  and  $y$  coordinate of its center. The desired value for these is set to the value of the corresponding attribute. The incremental parser will recognise that the circles and text form a final state since the circles and text are almost concentric. It will then add constraints that the circles and text have the

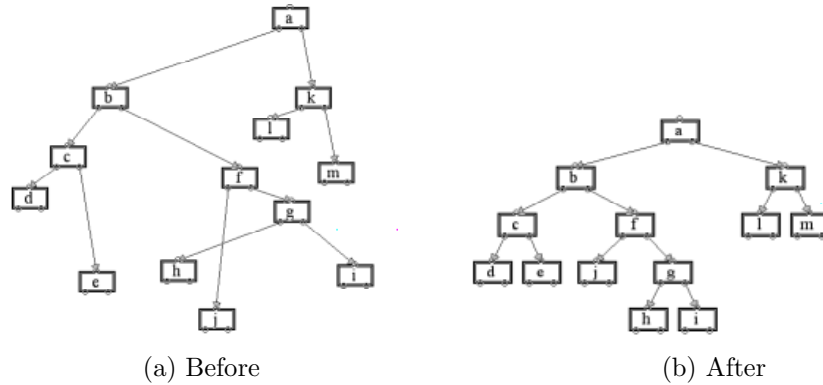


Figure 2. Beautification of a Binary Tree

same center and call `Solve` to determine new values for the `QCFloats` modelling the objects’ geometric attributes. `Solve` will find values that satisfy these constraints but which are as close as possible to the initial position. The editor then redraws the diagram with the new values for geometric attributes. Figure 1(b) shows the diagram after such error correction.

The second use of QOCA in Penguins is to support direct manipulation during editing of diagrams. Constraints inferred during parsing are maintained during manipulation, thus providing an editor which behaves as if it understands the semantics of the visual language. For instance, if a user moves any object in the final state shown in Figure 1(b), the other components will also move so as to maintain the constraint that all objects have the same center.

The third (and last) use of QOCA in Penguins is for diagram beautification such as, for instance, pretty printing a binary tree. This is achieved by adding visual language specific beautification constraints, such as parent nodes are centered between their children, solving the new system of constraints to determine new values for the objects’ geometric attributes, redrawing the diagram to reflect this beautification, and finally removing the beautification constraints. For instance, Figure 2 (taken from [8]) shows a binary tree before and after beautification.

One pleasing observation has been that linear equations and inequalities have proven sufficient to approximate the constraints in a wide variety of visual languages, namely, state transition diagrams, flow charts, mathematical equations, trees, and message sequence charts. However, in the case of state transition diagrams it has required some ingenuity to model the “arrow attached to circumference of state” relationship using linear constraints (the arrow is actually modelled as going from the center of the state with an invisible component). It would have been considerably more natural to use non-linear constraints. It was also only possible to handle limited forms of beautification. In particular beautification for graph-like visual languages such as state transition diagrams was rather restricted and could not, for instance, radically reorganise the diagram to minimise edge crossings.

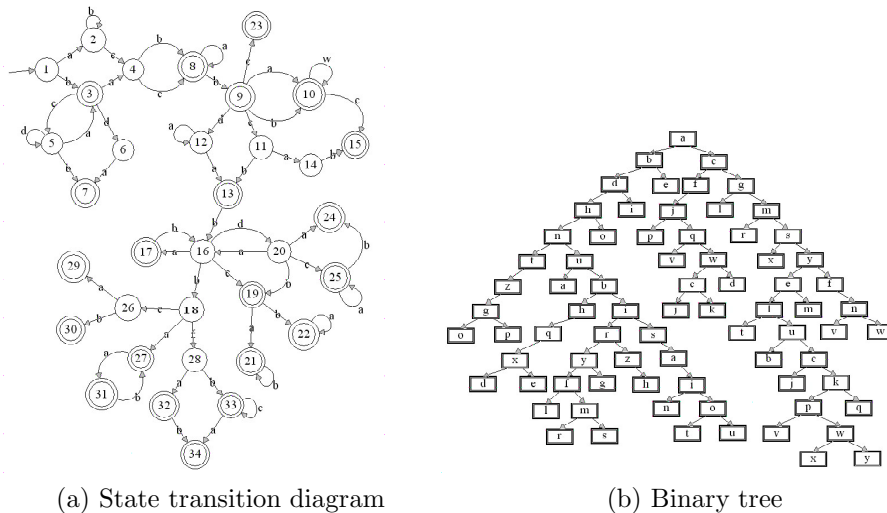


Figure 3. Sample Diagrams

Another application in which we have employed QOCA is for the dynamic layout of multiple column web documents composed of text, images and tables [4]. As the viewer changes font selection or resizes the browser, the layout of the document is modified. Geometric constraints ensure that the layout is appropriate. In this context linear equality and inequality constraints have proven powerful enough.

The final applications we have used QOCA for are in graph layout. As we have indicated above, currently QOCA is not suitable for arbitrary graph layout since this requires minimization of a complex non-quadratic objective function which captures aesthetics like reducing the number of edge crossings. However, QOCA has proven suitable for tree layout, in particular, rooted tree layout. Since it also allows arbitrary linear constraints on node placement it is more flexible than other approaches to tree layout. QOCA has also proven useful for quickly finding an initial graph layout, before using more expensive optimization techniques to improve the layout [10]. Finally, it has proven useful for removing node label overlapping from a given graph layout [11]. Almost all graph layout algorithms treat nodes as points. Thus when a graph with labelled nodes is drawn with the generated layout, the node labels may overlap. QOCA is used in a second phase to reposition nodes so that their labels do not overlap. Linear constraints are used to ensure that the structure of the original layout is preserved by dictating that the relative horizontal and vertical ordering of nodes remains the same and that node label overlapping is removed by adding a non-overlap constraint in either the horizontal or vertical direction for each pair of possibly overlapping nodes. By setting the desired values of node positions to be 0, the area of the new layout can be minimized.

The second aspect of QOCA requiring evaluation is the speed of constraint solving, in particular whether it is fast enough to be used in practical applications.

Our test data comes from two applications generated by the Penguins system. The first is an incremental parser for recognizing state transition diagrams and the associated graphics editor. The second is similar except that the visual language is that of binary trees. The first application only requires equalities, so we have used it to compare all three solvers, while in the second application inequality constraints are used to preserve the left-to-right ordering of the node's children and to ensure that parents are above their children. Thus we have used this to compare `QCLinIneqSolver` and `QCCassSolver`. We have tried QOCA on a variety of diagrams generating up to 1000 constraints. The most complex diagram in each test is shown in Figure 3.

We have timed the various calls to QOCA made during the incremental recognition of each of these diagrams and then during direct manipulation of a graphic object in the final diagram. In each application, after each graphic object is created by the user, the incremental parser is called. This calls `AddConstraint` to add constraints to the solver and also `Solve` to perform error correction. It may also call `RemoveConstraint` since adding new objects may invalidate earlier recognition of objects—for instance, a circle surrounding text may be recognized as a normal state, but if another concentric circle is added, the object must now be re-recognized as a final state. Direct manipulation with the graphic editor calls `SetEditVar` and `BeginEdit` to begin editing, `Resolve` during manipulation and `EndEdit` at the end of the direct manipulation. We have deliberately chosen diagrams which will exhibit worst-case behaviour of `Resolve` and `BeginEdit` since virtually all components of the diagram are connected and so almost all variables are dependent upon the edit variables.

All times are for a Dual Pentium-II 450 running Windows NT 4.0 and are in milliseconds. They are for the C++ version of QOCA; the Java version is considerably slower.

We have not given the times for `AddVar`, `RemoveVar` and `SetEditVar` since they are too small to be accurately measured. Times for the other operations are given in Table 1. We give the number of graphical components in the diagram (`Size`), the number of constraints (`C`) and the number of variables (`V`) generated by the application program (this does not include auxiliary constraints and variables generated within the solvers). For each operation and solver `QCLinEqSolver` (`Eq`), `QCLinIneqSolver` (`In`) and `QCCassSolver` (`Cs`) we give the average and then in brackets the maximum elapsed time taken to perform each operation of this kind when constructing the test diagram.

The performance of all three solvers is very satisfactory when adding or removing constraints. However we note that the application does not often remove constraints, so the data is sparse.

All three perform well when only equalities are used. `QCLinEqSolver` is slightly faster than `QCLinIneqSolver`, and `QCCassSolver` during `Resolve` and `EndEdit` but slower than `QCCassSolver` (because of the overhead of constructing a second tableau) in `Solve` and `BeginEdit`. `QCCassSolver` has significantly slower worst case performance in `AddConstraint`. We note that the speed difference between `QCLinIneqSolver` and `QCLinEqSolver` is anomalous since (when restricted to equa-

	Size	50	100	150	200	50	100	150
	C	248	508	776	1020	456	902	1356
	V	339	663	1010	1323	524	1008	1524
BeginEdit	Eq	3 (16)	12 (47)	31 (141)	53 (281)	- -	- -	- -
	In	3 (16)	11 (47)	23 (94)	36 (141)	13 (31)	141 (1140)	656 (2609)
	Cs	0 (0)	0 (16)	2 (16)	3 (16)	0 (0)	0 (0)	1 (16)
EndEdit	Eq	0 (0)	0 (0)	0 (0)	0 (0)	- -	- -	- -
	In	0 (0)	0 (0)	0 (15)	0 (16)	0 (0)	1 (15)	1 (15)
	Cs	0 (0)	0 (15)	2 (16)	5 (16)	0 (0)	2 (16)	5 (16)
Solve	Eq	11 (32)	38 (110)	80 (235)	147 (454)	- -	- -	- -
	In	7 (16)	22 (63)	43 (125)	71 (188)	22 (63)	97 (485)	348 (1250)
	Cs	8 (141)	13 (250)	18 (250)	24 (250)	5 (16)	9 (47)	15 (63)
Resolve	Eq	0 (0)	0 (16)	0 (16)	0 (16)	- -	- -	- -
	In	0 (16)	1 (16)	3 (16)	6 (32)	1 (16)	5 (63)	12 (141)
	Cs	0 (16)	1 (16)	2 (16)	3 (31)	1 (16)	3 (16)	4 (32)
Add-Constraint	Eq	0 (0)	0 (15)	0 (16)	0 (16)	- -	- -	- -
	In	0 (15)	0 (16)	0 (16)	0 (16)	0 (0)	0 (140)	0 (140)
	Cs	0 (172)	1 (484)	1 (484)	1 (484)	0 (16)	0 (16)	0 (16)
Remove-Constraint	Eq	0 (0)	0 (0)	0 (0)	0 (15)	- -	- -	- -
	In	0 (0)	0 (15)	0 (15)	0 (16)	0 (16)	0 (16)	0 (47)
	Cs	0 (0)	0 (16)	0 (16)	0 (16)	0 (16)	1 (16)	1 (16)

(a) State transition diagram application      (b) Binary tree application

Table 1. Speed of Constraint Solving

tions) `QCLinIneqSolver` uses essentially the same algorithm as `QCLinEqSolver`. We are currently looking into this.

When inequalities are used the performance of `QCLinIneqSolver` is acceptable for diagrams with less than 1000 constraints. For more complex diagrams the time taken for `Solve` and `BeginEdit` becomes noticeable in an interactive context.<sup>6</sup> The performance of `QCCassSolver` is impressive.

Thus, `QCLinEqSolver` is probably the method of choice for equalities, while `QCCassSolver` is definitely the method of choice if inequalities are required.

Interestingly, the choice of metric (Manhattan distance or square of the Euclidean distance) does not appear noticeable to users of the Penguins generated editors. We suspect that differences in the solution were subconsciously adjusted for by the users because of the immediate visual feedback provided during direct manipulation.

## 5. Conclusion and Future Work

We have introduced the QOCA toolkit, its underlying constraint solving algorithms and constraint interaction metaphor. Empirical evaluation shows that the interface is simple to use and well-suited for a wide variety of interactive graphical applications. It also demonstrates that all three solvers are fast enough to be used in practical interactive applications involving up to 1000 interdependent constraints.

An earlier version of QOCA [5] investigated the use of an active set method for solving convex quadratic programming problems. In the short-term we plan to

extend this version of QOCA to provide this. In the longer term we plan to extend QOCA to handle non-linear arithmetic constraints.

The QOCA toolkit may be downloaded from the QOCA website at

<http://www.csse.monash.edu.au/projects/qoca/>

It is distributed under the terms of the GNU General Public License.

## Acknowledgements

Many people have helped in the development of the various QOCA versions. They include (in chronological order) Richard Helm, Tien Huynh, John Vlissides, Toby Sargeant, Tania Armstrong, Andrew Kelly, Yi Xiao, Alan Finlay and Peter Moulder.

## Notes

1. Current affiliation: Expert Software Services Pty Ltd, Melbourne 3000, Australia
2. However, we note that implementations of the constraint hierarchy model often provide a way of calling the solver so that it can “compile” code to efficiently edit some given variables [1, 2].
3. This is for efficiency. If a variable’s weights need to be changed dynamically, then a new dummy variable can be added to the solver with the desired weights and an equality constraint equating it to the original variable added to the solver.
4. Actually, `Solve` comes in two flavours: `QuickSolve` and `CompleteSolve`. By default `QuickSolve` is called. It assumes that the application programmer has not changed the desired value for any `QCFloat` since the last time `Solve` or `Resolve` was called. `CompleteSolve` checks the desired values for all `QCFloats` in the solver and updates the values in the solver before solving.
5. Block addition of constraints and `ChangeRHS` is currently not provided by `QCCassSolver`.
6. However, it would be possible to perform these actions in background so as to provide acceptable performance for the user.

## References

1. A. Borning and B. 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.
2. A. Borning and B. Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints*, 3(1):9–32, 1998.
3. A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.
4. A. Borning, R. Lin, and K. Marriott. Constraints for the Web. In *Proceedings of the Fifth ACM International Multi-Media Conference*, pages 173–182, November 1997.
5. A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 10th ACM Symposium on User Interface Software and Technology*, pages 87–96, 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. S.S. Chok and K. Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of the 11th ACM Symposium on User Interface Software and Technology*, pages 185–194, 1998.

8. S.S. Chok, K. Marriott, and T. Paton. Constraint-based diagram beautification. In *Proceedings of the IEEE Symposium on Visual Languages (VL'99)*, pages 12–19, 1999.
9. R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, Chichester, 1987.
10. W. He and K. Marriott. Constrained graph layout. *Constraints*, 3(4):289–314, 1998.
11. W. He and K. Marriott. Removing node overlapping using constrained optimisation. In *Twenty-First Australasian Computer Science Conf.*, pages 169–180. Springer-Verlag, 1998.
12. R. Helm, T. Huynh, K. Marriott, and J. Vlissides. An object-oriented architecture for constraint-based graphical editing. In C. Laffra, E. Blake, V. de Mey, and X. Pintado, editors, *Object-Oriented Programming for Graphics*, pages 217–238. Springer-Verlag, 1995.
13. H. Hosobe, S. Matsuoka, and A. Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 237–251. Springer-Verlag LNCS 1118, 1996.
14. H. Hosobe, K. Miyashita, S. Takahashi, S. Matsuoka, and A. Yonezawa. Locally simultaneous constraint satisfaction. In *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, pages 51–62. Springer-Verlag LNCS 874, 1994.
15. T. Huynh and K. Marriott. Incremental constraint deletion in systems of linear constraints. *Information Processing Letters*, 55:111–115, 1995.
16. K. Marriott, S.S. Chok, and A. Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming (CP98)*, pages 340–354, 1998.
17. K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
18. I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.
19. Christopher J. Van Wyk. A class library for solving simultaneous equations. In *Proceedings of the USENIX C++ Technical Conference*, pages 229–234, 1991.
20. Christopher J. Van Wyk. Arithmetic equality constraints as C++ statements. *Software-Practice and Experience*, 22(6):467–494, 1992.