Solving Disjunctive Constraints for Interactive Graphical Applications

Kim Marriott[1], Peter Moulder[1], Peter J. Stuckey[2], and Alan Borning[3]

[1] School of Comp. Science & Soft. Eng., Monash University, Australia
[2] Dept. of Comp. Science & Soft. Eng., University of Melbourne, Australia
[3] Dept. of Computer Science & Eng., University of Washington, Seattle, USA

Abstract. In interactive graphical applications we often require that objects do not overlap. Such non-overlap constraints can be modelled as disjunctions of arithmetic inequalities. Unfortunately, disjunctions are typically not handled by constraint solvers that support direct manipulation, in part because solving such problems is NP-hard. We show here that is in fact possible to (re-)solve systems of disjunctive constraints representing non-overlap constraints sufficiently fast to support direct manipulation in interactive graphical applications. The key insight behind our algorithms is that the disjuncts in a non-overlap constraint are not disjoint: during direct manipulation we need only move between disjuncts that are adjacent in the sense that they share the current solution. We give both a generic algorithm, and a version specialised for linear arithmetic constraints that makes use of the Cassowary constraint solving algorithm.

1 Introduction

In many constraint-based interactive graphical applications, we wish to declare that several objects should not overlap. When reduced to arithmetic inequality constraints, this becomes a disjunction. As a motivating example, consider the diagram in Figure 1(a) of a 4×3 box and a 2×2 right triangle. The positions of the box and right triangle are given by the coordinates of the lower left-hand corners ((x_B,y_B) and (x_T,y_T)). A user editing this diagram might well want to constrain the box and triangle to never overlap. We can model this using a disjunction of linear constraints that represent the five (linear) ways we can ensure that non-overlapping holds. These are illustrated in Figure 1(b), and depict the five constraints

$$\begin{array}{l} x_T \geq x_B + 4 \;\;\vee\;\; y_T \geq y_B + 3 \;\;\vee\;\; y_T \leq y_B - 2 \;\;\vee \\ x_T \leq x_B - 2 \qquad\vee\qquad x_T + y_T \leq x_B + y_B - 2 \end{array}$$

During direct manipulation of, say, the triangle, the solver is allowed to move it to any location that does not cause overlap. For instance, the triangle can be moved around the box. However, if it is moved directly to the left, then once it touches the box, the box (assuming it is unconstrained) will also be pushed left to ensure that overlap does not occur.

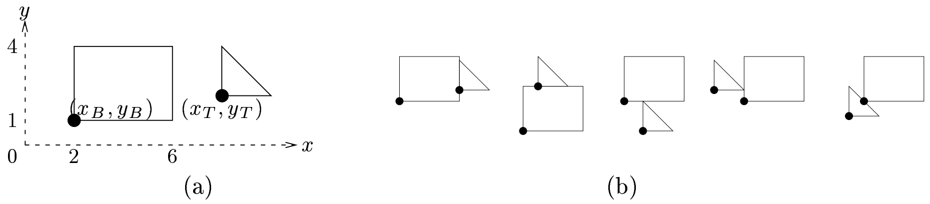


Fig. 1. Simple constrained picture, and five ways to ensure non-overlap.

Unfortunately, current constraint solving technology for interactive graphical applications cannot handle such disjunctive constraints. In part this is because solving such disjunctive systems is, in general, NP-hard. Thus, it seems very difficult to develop constraint solving algorithms that will be sufficiently fast for interactive applications and, in particular, support direct manipulation. An additional difficulty is that we wish to solve such disjunctive constraints in combination with the sort of constraints that are currently provided for interactive graphical applications. As an example, consider the state chart-like diagram shown in Figure 8(a).[1] In a constraint-based editor for such diagrams, we would like to combine non-overlap constraints with containment and connection constraints.

In this paper, we show that is in fact possible to (re-)solve systems of disjunctive constraints representing non-overlap constraints sufficiently fast to support direct manipulation in interactive graphical applications. The key insight behind our algorithms is that the disjuncts in a non-overlap constraint are not disjoint: during direct manipulation we need only move between adjacent disjuncts. At any given time one of the disjuncts will be active (and hence enforced by the solver). As we move to a new solution in which we must make one of the other disjuncts active instead, at the transition point the solution will satisfy both the current disjunct and the new one. This reflects that we want the graphical objects to behave sensibly and continuously during direct manipulation, and so we do not allow transitions through unsatisfiable regions, i.e., we do not allow objects to magically move through one another.

The paper includes three main technical contributions. The first is a general algorithm for solving such non-overlap constraint problems. The algorithm is generic in the choice of the underlying solver used to solve conjunctions of constraints. It is a true extension of the underlying solver, since it allows disjunctions in combination with whatever conjunctive constraints are provided by the underlying solver. We also show how the algorithm extends naturally to the case where the non-overlap constraints are preferred rather than required (Section 2). The second contribution is a specialisation of our generic algorithm to the case when the underlying solver is the Cassowary linear arithmetic constraint solver [3] (Sections 3 & 4). Cassowary is a simplex-based solver, and we can use the information in the simplex tableau to guide the search between disjuncts. Our final contribution is an empirical evaluation of this algorithm (Section 5).

---

[1] State charts, introduced by David Harel [7], are now part of the Unified Modelling Language [12], rapidly becoming the industry standard for object-oriented design.

We investigate both the speed of resolving and the expressiveness of disjunctions of linear constraints.

Starting with Sutherland [14], there has been considerable work on developing constraint solving algorithms for supporting direct manipulation in interactive graphical applications. These approaches fall into four main classes: propagation based (e.g. [13, 15]); linear arithmetic solver based (e.g. [3]); geometric solver-based (e.g. [4, 8]); and general non-linear optimisation methods such as Newton-Rapson iteration (e.g. [5]). However, none of these techniques support disjunctive constraints for modelling non-overlap. The only work that we know of that handles non-overlap constraints is that of Baraff [1], who uses a force based approach, modelling the non-overlap constraint between objects by a repulsion between them if they touch. Our approach differs in that it is generic and in that it handles non-overlap constraints in conjunction with other sorts of constraints. Subsequently, Harada, Witkin, and Baraff [6] extended the approach of [1] to support application-specific rules that allow temporary violation of non-overlap constraints in direct manipulation, so that the user can, if necessary, pass one object through another. Such application-specific rules could also be built on top of our algorithms.

2 A General Algorithm for Solving Disjunctions

We are interested in rapidly (re)-solving systems of constraints to support direct manipulation in interactive graphical applications. Graphical objects are displayed on the screen, with their geometric attributes represented by constrainable variables. Usually, the required constraints in such applications are not enough to uniquely fix a solution, i.e. the system of constraints is *underconstrained*. However, since we need to display a concrete diagram, the constraint solver must always determine an assignment θ to the variables that satisfies the constraints. Since we do not want objects to move unnecessarily over the screen, we prefer that the objects (and hence their attributes) stay where they are.

Such preferences can be formalised in terms of *constraint hierarchies* [2], one formalism for representing soft constraints. The idea is that constraints can have an associated strength that indicates to the solver how important it is to satisfy that constraint. There is a distinguished strength *required* which means that the constraint must be satisfied. By convention, constraints without an explicit strength are assumed to be required.

Given constraint hierarchies, it is simple to formalise the constraint solving required during direct manipulation. We have a conjunctive system of constraints C, some of which may be required, some of which may be not. We have some variables, typically one or two, say x and y, that correspond to the graphical attributes such as position that are being edited through direct manipulation. Let the remaining variables be $v_1, \ldots, v_n$ and let the current value of each v_i be a_i. The constraint solver must repeatedly resolve a system of form

$$C \wedge v_1 =_{stay} a_1 \wedge \cdots \wedge v_n =_{stay} a_n \wedge x =_{edit} b_1 \wedge y =_{edit} b_2.$$

```
disj_solve(C,active[])
    let C be of form C_0 ∧ D_1 ∧ ··· ∧ D_n where
    C_0 is a conjunction of constraints, and each D_i is of form
    𝒟_i^1 ∨ ··· ∨ 𝒟_i^{n_i}
    repeat
        θ := csolv(C_0 ∧ ⋀_{i=1}^n 𝒟_i^{active[i]})
        finished := true
        for i := 1,...,n do
            current := active[i]
            active[i] := dchoose(𝒟_i^1 ∨ ··· ∨ 𝒟_i^{n_i}, current, θ)
            if active[i] ≠ current then
                finished := false
                break  % Exit 'for' loop.
            endif
        endfor
    until finished
    return θ
```

Fig. 2. Generic algorithm for handling non-overlap constraints.

for different values of b_1 and b_2. The *stay constraints*, $v_1 =_{stay} a_1 \wedge \cdots \wedge v_n =_{stay} a_n$, indicate our preference that attributes are not changed unnecessarily, while the *edit constraints*, $x =_{edit} b_1 \wedge y =_{edit} b_2$, reflect our desire to give x and y the new values b_1 and b_2, respectively. Clearly the *edit* strength should be greater than the *stay* strength for editing to have the desired behaviour.

We can now describe our generic algorithm disj_solve for supporting direct manipulation in the presence of disjunctive constraints modelling non-overlap. It is given in Figure 2. It is designed to support rapid resolving during direct manipulation by being called repeatedly with different desired values for the edit variables. The algorithm is parametric in the choice of an underlying conjunctive constraint solver *csolv*. The solver takes a conjunction of constraints, including stay and edit constraints, and returns a new solution θ. The algorithm is also parametric in the choice of the function *dchoose* which chooses which disjunct in each disjunction is to be made active.

This algorithm is extremely simple. It takes a system of constraints C consisting of conjunctive constraints C_0 conjoined with disjunctive constraints $D_1, \ldots, D_n$, and an array *active* such that for each disjunction D_i, *active*$[i]$ is the index of the currently active disjunct in D_i. We require that the initial *active* value have a feasible solution. The algorithm uses *csolv* to compute the solution θ using the currently active disjunct in each disjunction. Then *dchoose* is called for each disjunction, to see if the active disjunct in that disjunction should be changed. If so, the process is repeated. If not, the algorithm terminates and returns θ. The algorithm is correct in that θ must be a solution of C since it is a solution of C_0 and one disjunct in each disjunction D_i. In practice, for efficiency *csolv* should use incremental constraint solving methods, since *csolv* is called repeatedly with a sequence of problems differing in only one constraint.

Clearly, the choice of *dchoose* is crucial to the efficiency and quality of solution found by disj_solve, since it guides the search through the various disjuncts. A bad choice of *dchoose* could even lead to looping and non-termination, unless some other provision is made. One simple choice for the definition of $dchoose(\mathcal{D}^1 \vee \cdots \vee \mathcal{D}^n, i, \theta)$ is to return j for some $j \neq i$ where θ is a solution of $\mathcal{D}^j$ and $\mathcal{D}^j$ has not been active before, or else i if no such j exists. A problem with this definition is that, even if a disjunction is irrelevant to the quality of solution, the algorithm may explore other disjuncts in the disjunction. We can improve this definition by only choosing a different disjunct from $\mathcal{D}^i$ if $\mathcal{D}^i$ is "active" in the sense that by removing it we could find a better solution. Another improvement is only to move to another disjunct if we can ensure that this leads to a better solution. Regardless, the key to the definition of *dchoose* is that it only chooses a j such that θ is a solution of $\mathcal{D}^j$. This greatly limits the search space and means that we use a hill-climbing strategy. Importantly, it means that we only move smoothly between disjuncts, giving rise to continuous, predictable behaviour during direct manipulation.

It is simple to modify the algorithm to handle the case of overlap constraints that are not required but rather are preferred with some strength w. We simply rewrite each such disjunction D_i to include an error variable for that disjunction e_i, and then conjoin the constraint $e_i =_w 0$ to C_0. For instance, if we prefer that the triangle and box from our motivating example do not overlap with strength *strong* then we can implement this using the constraints

$$\begin{array}{lll} & (e =_{strong} 0) & \\ \wedge & (& x_T \geq x_B + 4 + e \quad \vee \quad y_T \geq y_B + 3 + e \quad \vee \quad y_T \leq y_B - 2 + e \\ & \vee & x_T \leq x_B - 2 + e \quad \vee \quad x_T + y_T \leq x_B + y_B - 2 + e \quad) \end{array}$$

The only difficulty is that we need to modify *dchoose* to allow disjuncts to be swapped as long as the associated error does not increase.

It is instructive to consider the limitations of our approach. First, there is no guarantee that it will find the globally best solution. In the context of interactive graphical applications, this is not as significant a defect as it might appear. As long as direct manipulation behaves predictably, the user can search for the best solution interactively. Second, there is an assumption that disjuncts in a disjunction are not disjoint. This means that we cannot directly handle a "snap to grid" constraint such as $x = 1 \vee x = 2 \vee \cdots \vee x = n$ in which we require that position attributes can take only a fixed number of values, since there is no way to move between these disjuncts. (One way of handling such constraints is using integer programming techniques; see e.g. [11].)

3 Simplex Optimisation and the Cassowary Algorithm

We now give an instantiation of our generic algorithm for the case when the underlying solver is simplex based. We shall first review the simplex optimisation and the Cassowary Algorithm.

The simplex algorithm takes a conjunction of linear arithmetic constraints C and a linear arithmetic objective function f which is to be minimised. These must

simplex(C, f, $\underline{active[]}$)
repeat
let f have form $h + \sum_{j=1}^{m} d_j y_j$ and
let C have form $\bigwedge_{i=1}^{n} x_i = k_i + \sum_{j=1}^{m} a_{ij} y_j$
% Choose variable y_J to become basic.
if $\forall [j \in \{1, \ldots, m\}]$ ($d_j \geq 0$ $\underline{\text{or } \exists i . y_j \in active[i]}$) **then**
return $\underline{\text{(C, f)}}$ % An optimal solution has been found.
endif
choose $J \in \{1, \ldots, m\}$ such that $d_J < 0$ $\underline{\text{and } \forall i . y_j \notin active[i]}$
% Choose variable x_I to become non-basic
choose $I \in \{1, \ldots, n\}$ such that
$-k_I / a_{IJ} = \min \{-k_i / a_{iJ} \mid i \in \{1, \ldots, n\} \text{ and } a_{iJ} < 0\}$
$e := (x_I - k_I - \sum_{j=1, j \neq J}^{m} a_{Ij} y_j) / a_{IJ}$
$C[I] := (y_J = e)$
replace y_J by e in f
for each $i \in \{1, \ldots, n\}$
if $i \neq I$ **then** replace y_J by e in $C[i]$ **endif**
endfor
endrepeat

Fig. 3. Simplex optimization.

be in *basic feasible solved form*. More exactly, f should have form $h + \sum_{j=1}^{m} d_j y_j$ and C should have form $\bigwedge_{i=1}^{n} x_i = k_i + \sum_{j=1}^{m} a_{ij} y_j$. The variables $y_1, \ldots, y_m$ are called *parameters*, while the variables $x_1, \ldots, x_n$ are said to be *basic*. All variables are implicitly required to be non-negative, and the right-hand side constants (the k_i's) are required to be non-negative.[2] Although the constraints are equations, linear inequalities can be handled by adding a *slack* variable and transforming to an equation. Any set of constraints in basic feasible solved form has an associated variable assignment, which, because of the definition of basic feasible solved form, must be a solution of the constraints. In the case of C above it is

$$\{x_1 \mapsto k_1, \ldots, x_n \mapsto k_n, y_1 \mapsto 0, \ldots, y_m \mapsto 0\}.$$

The Simplex Algorithm is shown in Figure 3, and takes as inputs the simplex tableau C and the objective function f. The underlined text in the algorithm should be ignored for now. The algorithm repeatedly selects 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 a parameter and we want to make it basic.) Pivoting on such a variable cannot increase the value of the objective function (and usually decreases it). If no such variable exists, the optimum has been reached. Next we determine the exit variable x_I. We must choose this variable so that it maintains basic feasible solved form by ensuring that the new k_i's are still positive after pivoting. That is, we must choose an I so that $-k_I / a_{IJ}$ is a minimum element of the set

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

[2] See e.g. [3] for efficient handling of unrestricted-in-sign variables.

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

One obvious issue is how we convert a system of equations into basic feasible solved form. Luckily the Simplex Algorithm itself can be used to do this. An incremental version of this algorithm is described in [10]. The only point to note is that adding a new constraint may require that simplex optimisation must be performed.

In the special case that we have constraints in a basic solved form which is infeasible in the sense that some right-hand side constants (the k_is) may be negative, but which is optimal in the sense that all coefficients in the objective function are non-negative, we can use the Dual Simplex Algorithm to restore feasibility. This is similar to the Simplex Algorithm, except that the role of the objective function and the right-hand side constants are reversed.

The Simplex Algorithm and the Dual Simplex Algorithm provide a good basis for fast incremental resolving of linear arithmetic constraints for interactive graphical applications. One simplex-based algorithm for solving direct manipulation constraints is Cassowary [3]. The key idea behind the approach is to rewrite non-required constraints (such as edit and stay constraints) of form $x =_w k$ into $x + \delta_x^+ - \delta_x^- = k$ and add the term $c_w \times \delta_x^+ + c_w \times \delta_x^-$ to the objective function, where δ_x^+ and δ_x^- are error variables, and c_w is a coefficient reflecting the strength w. The Dual Simplex Algorithm can now be used to solve the sequence of problems arising in direct manipulation, since only the right hand side constants are changing.

4 A Disjunctive Solver Based on the Cassowary Algorithm

We now describe how to embed the Cassowary Algorithm into the generic algorithm given earlier. We could embed it directly by simply using it as the constraint solver *csolv* referenced in Figure 2, but we can do better than this. It is moderately expensive to incrementally add and delete constraints using the simplex method. For this reason, we keep all disjuncts in the solved form, rather than moving them in and out of the solved form whenever we switch disjuncts. Since we only want one disjunct from each disjunction to be active at any time, we represent each disjunction using linear constraints together with error variables representing the degree of violation of each disjunct. As long as one error variable in the disjunction has value zero, the disjunctive constraint is satisfied. (The other error variables can be disregarded.)

More formally, the *error form* of an equation $a_1x_1 + \ldots + a_nx_n = b$ is $a_1x_1 + \ldots + a_nx_n + e^- - e^+ = b$ where e^+ and e^- are two non-negative *error variables*, representing the degree to which the equation is satisfied, while the *error form* of an inequality $a_1x_1 + \ldots + a_nx_n \leq b$ is $a_1x_1 + \ldots + a_nx_n + s - e = b$ where s is the slack variable and e is the *error variable*. Both s and e must be non-negative. Note that for any values of $x_1, \ldots, x_n$ there is a solution of the error form of each linear constraint. Note also that if we constrain the error variables for some linear constraint c to be zero, then the error form of c is equivalent to c.

The *conjunctive version* of a disjunctive constraint D is the conjunction of the error forms of the disjuncts $\mathcal{D}^1, \ldots, \mathcal{D}^n$ in D. The conjunctive version of a disjunctive constraint D does not ensure that D is satisfied. In order to ensure that the disjunctive constraint is satisfied we must ensure that, for some disjunct $\mathcal{D}^i$ in D, the error variable(s) of the error form of $\mathcal{D}^i$ take value 0.

The conjunctive version of our example disjunctive constraint is

$$\begin{array}{l} x_T + e_1 = x_B + 4 + s_1 \quad \wedge \quad y_T + e_2 = y_B + 3 + s_2 \quad \wedge \quad y_T + s_3 = y_B - 2 + e_3 \\ \wedge \qquad x_T + s_4 = x_B - 2 + e_4 \qquad \wedge \qquad x_T + y_T + s_5 = x_B + y_B - 2 + e_5 \end{array}$$

where the error variables $e_1, \ldots, e_5$ and slack variables $s_1, \ldots, s_5$ are required to be non-negative. As long as one of the error variable takes value zero in a solution, then it is a solution of the original non-overlap constraint. A solution (corresponding to Figure 1(a)) is

$$\begin{array}{l} \{x_B \mapsto 2,\, y_B \mapsto 1,\, x_T \mapsto 8,\, y_T \mapsto 2,\, s_1 \mapsto 2,\, s_2 \mapsto 0,\, s_3 \mapsto 0,\, s_4 \mapsto 0,\, s_5 \mapsto 0, \\ e_1 \mapsto 0,\, e_2 \mapsto 2,\, e_3 \mapsto 3,\, e_4 \mapsto 8,\, e_5 \mapsto 9\} \end{array}$$

We must modify the Simplex Algorithm shown in Figure 3 to ensure that the error variable from the active disjunct in each disjunction is always kept zero. The changes are shown as underlined text in the figure. They are rather simple: we ensure that such *active error variables* are always kept as parameters and are never chosen to become basic. Thus, we must pass an extra argument to the Simplex Algorithm, namely *active*, the array of currently active error variables. For each disjunction D_i, *active*$[i]$ is the set of active error variables in the error form of the the active disjunct in D_i. Note that *active*$[i]$ will contain one variable if the disjunct is an inequality and two if it is an equation. When choosing the new basic variable y_J, we ignore any active error variables in the objective function: they cannot be chosen to become basic, and are allowed to have a negative coefficient in the objective function. We can modify the Dual Simplex Algorithm similarly.

The generic algorithm can be readily specialised to call the modified Simplex and Dual Simplex Algorithms. It is shown in Figure 4. The algorithm takes the current basic feasible solved form of the constraints C and objective function f as well as the active error variables. The main syntactic differences between the generic algorithm and this specialised algorithm result from the need to call the Dual Simplex Algorithm rather than the Simplex Algorithm when the algorithm is first entered. This is because we assume that only the right-hand side constants have been modified as the result of changing the desired values for the edit variables. (See [3] for further details.)

```
disj_simplex_solve(C, f, active[])
    (C, f) := dual_simplex(C, f, active)
    tried := ∅
    repeat
        let θ be the solution corresponding to C
        prevf := θ(f)
        finished := true
        let C be of form C_0 ∧ D_1 ∧ ··· ∧ D_n where
        C_0 is a conjunction of constraints and
        each D_i is the conjunctive version of a disjunction
        for i := 1, ..., n do
            current := active[i]
            active[i] := simplex_choose(D_i, current, f, θ, tried)
            if active[i] ≠ current then
                tried := tried ∪ active[i]
                if some y ∈ active[i] is basic then make y a parameter endif
                finished := false
                (C, f) := simplex(C, f, active)
                if θ(f) < prevf then tried := ∅ endif
                break  % Exit 'for' loop.
            endif
        endfor
    until finished = true
    return θ

simplex_choose(D, current, f, θ, tried)
    let D be of form c_1 ∧ ··· ∧ c_m
    if ∃y ∈ current s.t. y has a negative coefficient in f then
        for i := 1, ..., m do
            let E be the error variables in c_i
            if ∀[e ∈ E]. θ(e) = 0 and e ∉ tried and e ≠ current then
                return E
            endif
        endfor
    endif
    return current
```

Fig. 4. Algorithm for handling non-overlap constraints in the linear case.

The function simplex_choose for switching between disjuncts uses information in the objective function in the solved form to assist in the choice of disjunct. The coefficient of an active error variable in the objective function provides a heuristic indication of whether it would be advantageous to switch away from that disjunct: we try to switch away only if the coefficient is negative, indicating that the objective function value can be decreased by making the variable basic (if it results in a non-zero value). When choosing which disjunct to switch to, the value of the error variables in the inactive disjuncts indicate which disjuncts are satisfied by the current solution. This is another advantage of keeping the error form of all disjuncts in the solved form.

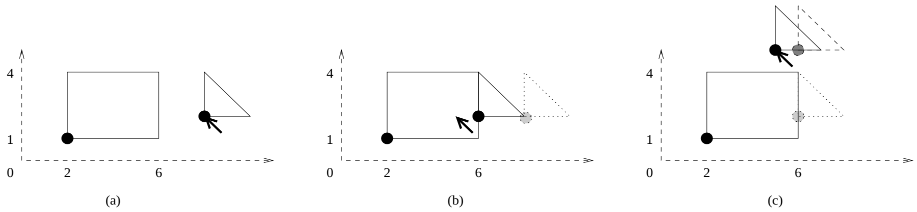


Fig. 5. Motion of the triangle during the mouse movement.

If a newly chosen active variable is basic, we first make it a parameter before re-optimizing. If an error variable is basic and takes value 0 in the current solution, the right-hand side constant must be 0. This means that we can pivot on any parameter other than active error variables in the solved form, making that parameter basic. For example, if the disjunct is an inequality, it is always possible to pivot on the slack variable. The only difficult case is if the left hand side in the solved form consists entirely of active disjunctive error variables. The simplest way of handling this case is to split the equality into two inequalities, thus ensuring that each of the two rows has a slack variable that can be made basic.

The algorithm maintains a set of *tried* active error variables, which is reset to empty whenever we improve the objective function. This prevents us looping infinitely trying different combinations of active constraints without improving the solution.

To illustrate the operation of the algorithm consider our running example. For simplicity let us fix the position of the box at (2,1) and add constraints that the triangle attempt to follow the mouse position (x_M, y_M). Using the Cassowary encoding, we add the edit constraints

$$x_T = x_M + \delta_x^+ - \delta_x^- \qquad \wedge \qquad y_T = y_M + \delta_y^+ - \delta_y^-$$

where $\delta_x^+, \delta_x^-, \delta_y^+, \delta_y^- \geq 0$, and minimise the objective function $\delta_x^+ + \delta_x^- + \delta_y^+ + \delta_y^-$ where for simplicity we assume that the coefficient for the edit strength is 1.0. The motion of the triangle is illustrated in Figure 5, with the mouse pointer indicated by an arrow. When there is a change of active constraints, the intermediate point is shown as a dashed triangle.

We assume the mouse begins at (8,2), the initial position of the triangle, and the initial basic feasible solved form is

$$\begin{array}{l}
\text{minimize } \delta_x^+ + \delta_x^- + \delta_y^+ + \delta_y^- \\
\begin{array}{lllllll}
x_T & = 8 & +\delta_x^+ & -\delta_x^- & & & \\
y_T & = 2 & +\delta_y^+ & -\delta_y^- & & & \\
s_1 & = 2 & +\delta_x^+ & -\delta_x^- & +e_1 & & \\
e_2 & = 2 & -\delta_y^+ & +\delta_y^- & +s_2 & & \\
e_3 & = 3 & +\delta_y^+ & -\delta_y^- & +s_3 & & \\
e_4 & = 8 & +\delta_x^+ & -\delta_x^- & +s_4 & & \\
e_5 & = 9 & +\delta_x^+ & -\delta_x^- & +\delta_y^+ & -\delta_y^- & +s_5
\end{array} \\
\qquad \qquad active: \; e_1
\end{array}$$

$$
\begin{array}{llllllll}
\multicolumn{8}{c}{\text{minimize } \delta_x^+ + \delta_x^- + \delta_y^+ + \delta_y^-} \\
x_T & = 5 & +\delta_x^+ & -\delta_x^- & & & & \\
y_T & = 2 & +\delta_y^+ & -\delta_y^- & & & & \\
s_1 & = -1 & +\delta_x^+ & -\delta_x^- & +e_1 & & & \\
e_2 & = 2 & -\delta_y^+ & +\delta_y^- & +s_2 & & & \\
e_3 & = 3 & +\delta_y^+ & -\delta_y^- & +s_3 & & & \\
e_4 & = 5 & +\delta_x^+ & -\delta_x^- & +s_4 & & & \\
e_5 & = 6 & +\delta_x^+ & -\delta_x^- & +\delta_y^+ & -\delta_y^- & +s_5 & \\
\multicolumn{8}{c}{\mathit{active}:\ e_1}
\end{array}
$$

(a)

$$
\begin{array}{lllllll}
\multicolumn{7}{c}{\text{minimize } 1 + s_1 - e_1 + 2\delta_x^- + \delta_y^+ + \delta_y^-} \\
x_T & = 6 & +s_1 & -e_1 & & & \\
y_T & = 2 & +\delta_y^+ & -\delta_y^- & & & \\
\delta_x^+ & = 1 & +\delta_x^- & +s_1 & -e_1 & & \\
e_2 & = 2 & -\delta_y^+ & +\delta_y^- & +s_2 & & \\
e_3 & = 3 & +\delta_y^+ & -\delta_y^- & +s_3 & & \\
e_4 & = 6 & +s_1 & -e_1 & +s_4 & & \\
e_5 & = 7 & +s_1 & -e_1 & +\delta_y^+ & -\delta_y^- & +s_5 \\
\multicolumn{7}{c}{\mathit{active}:\ e_1}
\end{array}
$$

(b)

$$
\begin{array}{lllllll}
\multicolumn{7}{c}{\text{minimize } 1 + s_1 - e_1 + 2\delta_x^- + \delta_y^+ + \delta_y^-} \\
x_T & = 6 & +s_1 & -e_1 & & & \\
y_T & = 5 & +\delta_y^+ & -\delta_y^- & & & \\
\delta_x^+ & = 1 & -\delta_x^- & +s_1 & -e_1 & & \\
s_2 & = 1 & +\delta_y^+ & -\delta_y^- & +e_2 & & \\
e_3 & = 6 & +\delta_y^+ & -\delta_y^- & +s_3 & & \\
e_4 & = 6 & +s_1 & -e_1 & +s_4 & & \\
e_5 & = 10 & +s_1 & -e_1 & +\delta_y^+ & -\delta_y^- & +s_5 \\
\multicolumn{7}{c}{\mathit{active}:\ e_1}
\end{array}
$$

(c)

$$
\begin{array}{lllllll}
\multicolumn{7}{c}{\text{minimize } \delta_x^+ + \delta_x^- + \delta_y^+ + \delta_y^-} \\
x_T & = 5 & +\delta_x^+ & -\delta_x^- & & & \\
y_T & = 5 & +\delta_y^+ & -\delta_y^- & & & \\
e_1 & = 1 & -\delta_x^+ & +\delta_x^- & +s_1 & & \\
s_2 & = 1 & +\delta_y^+ & -\delta_y^- & +e_2 & & \\
e_3 & = 6 & +\delta_y^+ & -\delta_y^- & +s_3 & & \\
e_4 & = 5 & +\delta_x^+ & -\delta_x^- & +s_4 & & \\
e_5 & = 9 & +\delta_x^+ & -\delta_x^- & +\delta_y^+ & -\delta_y^- & +s_5 \\
\multicolumn{7}{c}{\mathit{active}:\ e_2}
\end{array}
$$

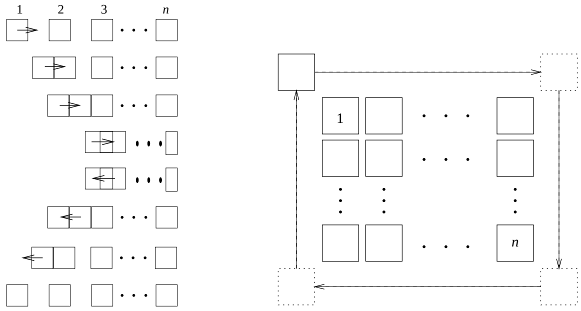
(d)

Fig. 6. Tableaus resulting during the edits of Figure 5.

This corresponds to the position in Figure 5(a). The special entry *active*: e_1 indicates that e_1 is an active error constraint and so is not allowed to enter the basis.

Suppose now we move the mouse to (5,2). The modified solved form is shown in Figure 6(a). We call disj_simplex_solve, which calls the dual_simplex algorithm. Since the solved form is no longer feasible, but still optimal, the Dual Simplex Algorithm recovers feasibility by performing a pivot that removes s_1 from the basis and enters δ_x^+ into the basis. This gives the tableau in Figure 6(b), whose corresponding solution gives position (6,2) for the triangle, illustrated in Figure 5(b). We now call simplex_choose for the single disjunction in the original constraint set. The appearance of $-e_1$ in the objective function means that a better solution could be found if we allowed e_1 to enter the basis, and so if possible we should switch disjuncts. However, since no other error variables are zero, we cannot switch disjuncts. Thus simplex_choose returns $\{e_1\}$, and, since the active error variables have not changed, disj_simplex_solve returns with this solved form.

Now the user moves the mouse to (5,5). The solved form is modified, giving a infeasible optimal solution. The call to disj_simplex_solve calls dual_simplex. This time we have e_2 as the exit variable and s_2 as the entry variable, resulting the tableau shown in Figure 6(c). Now we have a corresponding optimal solution positioning the triangle at (6,5) (the dashed triangle in Figure 5(c)) for this choice of active disjuncts. We call simplex_choose for the single disjunction in the constraint set. Again the appearance of $-e_1$ in the objective function means



(a) (b)

Fig. 7. Experiments for (a) overhead and (b) performance of disjunctive solving.

that a better solution could be found if we allowed e_1 to enter the basis, and so if possible we should switch disjuncts. This time, since e_2 is now a parameter, it takes value zero in the current solution, so we can make this disjunct active. Thus simplex_choose returns $\{e_2\}$. We therefore make this the active error variable and call simplex to optimise with respect to this new disjunct. It performs one pivot, with entry variable e_1 and exit variable δ_x^+, giving the tableau in Figure 6(d). Notice how we have moved to position (5,5) and changed which of the disjuncts is active (the final position in Figure 5(c)). Now, since there are no active error variables in the objective function, simplex_choose does not switch disjuncts and so disj_simplex_solve returns with the solution corresponding to the solved form.

5 Empirical Evaluation

In this section we provide a preliminary empirical evaluation of disj_simplex_solve. Our implementation is based on the C++ implementation of the Cassowary Algorithm in the QOCA toolkit [9]. All times are in milliseconds measured on a 333MHz Celeron-based computer. (Granularity of maximum re-solve times is 10ms.)

Our first experiment compares the overhead of disj_simplex_solve with the underlying Cassowary Algorithm. Figure 7(a) shows n boxes in a row with a small gap between them. Each box has a desired width but can be compressed to half of this width. The rightmost box has a fixed position. The others are free to move, but have stay constraints tending to keep them at their current location. For the disj_simplex_solve version of the problem we add a non-overlap constraint between each pair of boxes. In the Cassowary version of the experiment there is a constraint to preserve non-overlap of each pair of boxes by keeping their current relative position in the x direction. This corresponds to the active constraints chosen in the disj_simplex_solve version.

The experiment measures the average and maximum time required for a resolve during the direct manipulation scenario in which the leftmost box is moved as far right as possible, squashing the other boxes together until they all shrink to half width, and then moved back to its original position. The results

	Cassoway			disj_simplex_solve		
n	Cons	AveR	MaxR	Cons	AveR	MaxR
20	190	1	10	760	6	20
40	780	3	10	3120	31	90
60	1770	7	20	7080	86	250
80	3160	12	30	12640	184	530

(a)

n	Cons	Swaps	AveR	MaxR
200	800	1.4	7	50
400	1600	2.6	18	90
600	2400	3.7	31	130
800	3200	4.7	47	170
1000	4000	5.5	66	230
1200	4800	6.3	85	260

(b)

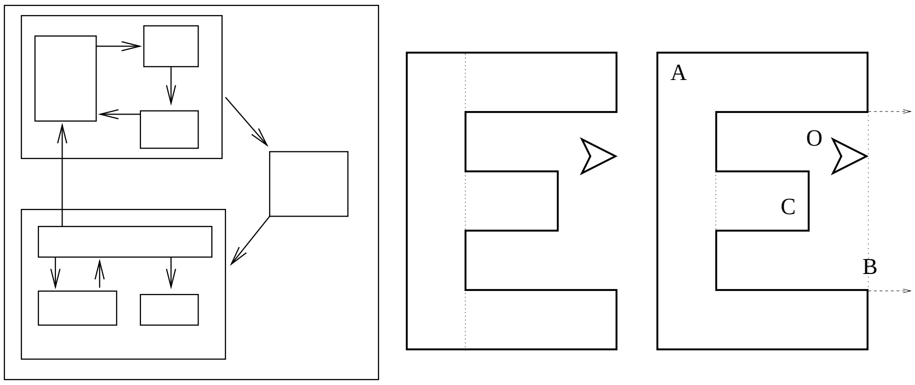
Table 1. Results for (a) overhead and (b) disjunctive swap speed.

shown in Table 1(a) gives the number n of boxes, for each version the number of linear constraints (Cons) in the solver, the average time (AveR), and maximum time (MaxR) to resolve during the direct manipulation (in milliseconds). Note that in this experiment no disjuncts change status from active to inactive or vice versa. The results show that there is a surprising amount of overhead involved in keeping non-active disjuncts in the solved form. We are currently investigating why: even with the same number of constraints in the solved forms, the original Cassowary seems significantly faster.

Our second experiment gives a feel for the performance of disj_simplex_solve when disjunct swapping takes place. Figure 7(b) shows n fixed size boxes arranged in a rectangle and a single box on the left-hand side of this collection. There is a non-overlap constraint between this box and each box in the collection. The experiment measures the average and maximum time required for a resolve during the direct manipulation scenario in which the isolated box is moved around the rectangle of boxes, back to its original position. Table 1(b) gives the number n of boxes, the number of linear constraints in the solver, the average and maximum time for each resolve, and the average number of disjunct swaps in each resolve. The results here show that disj_simplex_solve is sufficiently fast for supporting direct manipulation for systems of up to 5000 constraints and disjuncts.

Our third and fourth experiments give a feel for the expressiveness of disjunctions of linear constraints. In the third experiment we use the solver to model the constraints in the state chart-like diagram shown in Figure 8(a). It has non-overlap constraints between boxes in the same box, and containment constraints between boxes and their surrounding box. This gives rise to 20 linear constraints. For such a small number of constraints, re-solve time is negligible (0.04ms average; the maximum is not accurately measurable).

In the fourth experiment we demonstrate non-overlap with non-convex polygons. One way of modelling this is as simple convex polygons whose sides are "glued" together using constraints. Dotted lines in the Figure 8(b) show a simple convex decomposition of the E, requiring 24 linear constraints plus 4 disjunctions. However, one can model the situation using fewer constraints by allowing disjuncts to be conjunctions, perhaps even containing other disjunctions. Figure 8(c) illustrates the embedded-conjunction approach, which uses 12 linear



(a) (b) (c)

Fig. 8. Experiments to demonstrate expressiveness of disjunctive linear constraints.

constraints plus 2 disjunctions, implicitly defining the relation between the small "chevron" object O and the three objects A (the bounding box of E), B (the open sided rectangular gap in the E) and C (the middle bar of E), modelling the non-overlap of the E and O as

$$nonoverlap(O, A) \vee (inside(O, B) \wedge nonoverlap(O, C)).$$

In the test case, we have 8 "E" shapes and one "chevron" shape, all constrained to lie within a screen rectangle and constrained not to overlap each other. This yields 226 linear constraints and 36 disjunctions. The test case movements were constructed by manually dragging the shapes about each other, bumping corners against each other as much as possible. There were on average 0.3 disjunct swaps per re-solve. The average re-solve time was 0.6ms; the maximum was 20ms.

6 Conclusions

We have described an algorithm for rapidly resolving disjunctions of constraints. The algorithm is designed to support direct manipulation in interactive graphical applications which contain non-overlap constraints between graphical objects. It is generic in the underlying (conjunctive) constraint solver. We also give a specialisation of this algorithm for the case when the underlying constraint solver is the simplex-based linear arithmetic constraint solver, Cassowary.

Empirical evaluation of the Cassowary-based disjunctive solver is very encouraging, suggesting that systems of up to five thousand constraints can be solved in less than 100 milliseconds. We have also demonstrated that the solver can support non-overlap of complex non-convex polygons, and complex diagrams such as State Charts that contain non-overlap as well as containment constraints.

However, our experimental results indicate that keeping inactive disjuncts in the solved form has significant overhead. Thus, we intend to investigate a "dynamic" version of the Cassowary-based disjunctive solver in which disjuncts are only placed in the solver when they become active. Preliminary investigation by Nathan Hurst is very promising.

Acknowledgements

This research has been funded in part by an Australian ARC Large Grant A49927003, and in part by U.S. National Science Foundation Grant No. IIS-9975990. We thank Nathan Hurst for his insightful comments and criticisms.

References

1. David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94 Conference Proceedings*, pages 23–32. ACM, 1994.
2. Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.
3. Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, October 1997.
4. Ioannis Fudos. *Geometric Constraint Solving*. PhD thesis, Purdue University, Department of Computer Sciences, 1995.
5. Michael Gleicher. *A Differential Approach to Constraint Satisfaction*. PhD thesis, School of Computer Science, Carnegie-Mellon University, 1994.
6. Mikako Harada, Andrew Witkin, and David Baraff. Interactive physically-based manipulation of discrete/continuous models. In *SIGGRAPH '95 Conference Proceedings*, pages 199–208, Los Angeles, August 1995. ACM.
7. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
8. Glenn Kramer. A geometric constraint engine. *Artificial Intelligence*, 58(1–3):327–360, December 1992.
9. 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.
10. Kim Marriott and Peter Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.
11. George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.
12. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
13. Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.
14. Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Department of Electrical Engineering, MIT, January 1963.
15. Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.