# Dynamic Approximation of Complex Graphical Constraints by Linear Constraints

*Nathan Hurst* *Kim Marriott* *Peter Moulder*
School of Computer Science and Software Engineering
Monash University
Clayton, Victoria 3168, Australia
{njh,marriott,pmoulder}@mail.csse.monash.edu.au

### ABSTRACT

Current constraint solving techniques for interactive graphical applications cannot satisfactorily handle constraints such as non-overlap, or containment within non-convex shapes or shapes with smooth edges. We present a generic new technique for efficiently handling such kinds of constraints based on trust regions and linear arithmetic constraint solving. Our approach is to model these more complex constraints by a dynamically changing conjunction of linear constraints. At each stage, these give a local approximation to the complex constraints. During direct manipulation, linear constraints in the current local approximation can become *active* indicating that the current solution is on the boundary of the trust region for the approximation. The associated complex constraint is notified and it may choose to modify the current linear approximation. Empirical evaluation demonstrates that it is possible to (re-)solve systems of linear constraints that are dynamically approximating complex constraints such as non-overlap sufficiently quickly to support direct manipulation in interactive graphical applications.

**KEYWORDS:** Constraint-solving, direct manipulation, trust regions, linearization of constraints, non-overlap, containment.

### INTRODUCTION

Despite decades of research into specialised constraint-solving algorithms for interactive graphical applications, current constraint solving algorithms are still not powerful enough to solve constraints arising in several important types of applications. For instance, consider a constraint-based editor for state chart-like diagrams or a document editor allowing constraint-based placement of floating figures in a page. In both cases, we want to support direct manipulation of arbitrary diagram components in the context of non-overlap, containment, alignment and distribution constraints. Unfortunately, virtually all existing approaches to constraint solving for interactive graphical applications cannot handle the kind of (possibly cyclic) multi-way constraints arising in the above scenarios.

Here we present a generic new technique for efficiently handling such kinds of constraints based on linear arithmetic constraint solving. Simplex-based algorithms for incremental solving of conjunctions of linear arithmetic inequality constraints are very efficient and reasonably well understood [3, 11]. However, such solvers cannot directly handle the above sort of constraints because when couched in terms of linear constraints they are inherently disjunctive in nature. Our key idea is very simple: we model these more complex constraints by a dynamically changing conjunction of linear constraints. At each stage, the linear constraints give a local approximation to the constraints. Associated with the approximation are tests indicating for a given solution whether the approximation should be updated (e.g. whether the approximation is correct in the neighbourhood of the new solution found by the inner solver). During direct manipulation, linear constraints in the current local approximation can become *active*, which loosely means that removing this constraint might allow a solution closer to the user's desired solution. The associated complex constraint is notified and it may choose to modify the current linear approximation allowing a better solution to be found.

As a simple example, consider the two circles shown in Figure 1(a). We might want to constrain these so that they cannot overlap. This constraint is not expressible using a fixed conjunction of linear constraints. However, we can approximate the non-overlap constraint by the linear constraint that the smaller circle lies to the left of the vertical line shown in the figure. Now imagine that we wish to move the smaller circle along the path shown by the dashed arc in Figure 1(b). Clearly, this movement does not violate the non-overlap constraint but does violate the linear approximation. Our approach

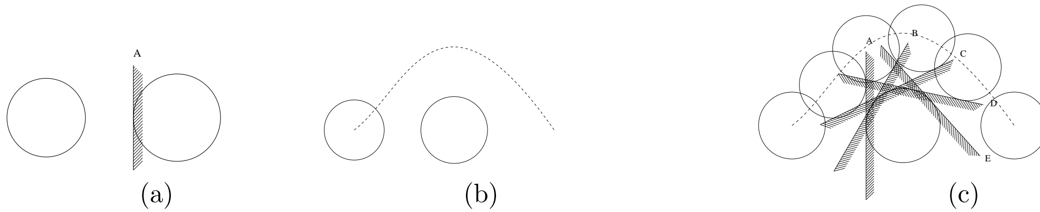


Figure 1: Dynamic linear approximation of non-overlap constraint for two circles.

handles this as shown in Figure 1(c). We can move the circle along the arc until it touches line A. At this point the inequality constraint corresponding to line A becomes active. The non-overlap constraint is notified and changes the linear approximation by removing the constraint that the small circle is to the left of A and adding the linear constraint that the circle is now required to be to the left of line B. As the user continues to move the circle along the arc, the inequality corresponding to line B will become active and the non-overlap constraint will be notified again, this time changing the approximation to the constraint that the smaller circle must be above line C. In turn the corresponding inequality to C will become active, the non-overlap constraint notified and the approximation changed to require the small circle to lie above line D. This is repeated again, with the inequality corresponding to line D made active and then replaced by the linear constraint that the small circle is to the right of line E. The circle can now be moved to the desired position. Notice that at all times the linear approximation is safe in the sense that it ensures that the non-overlap constraint is maintained. The important point is that because at each stage we are using a linear approximation, we can use additional linear constraints to express alignment, distribution, containment etc., and these will all work smoothly with the non-overlap constraint.

We have been pleasantly surprised with how effective dynamic linear approximation is. We shall see that it is possible to (re-)solve systems with complex constraints, such as non-overlap constraints, sufficiently fast to support direct manipulation in interactive graphical applications. In hindsight perhaps this is not too surprising: after all, linear approximation is the basis for much applied mathematics and many approaches to non-linear constrained optimisation [4].

The paper has four main technical contributions. The first is a generic algorithm for solving complex non-linear constraints by dynamically approximating the constraints with linear constraints. Our second technical contribution is to demonstrate the power of this approach by showing how this algorithm can be instantiated to handle non-overlap and containment constraints. However, if one has many mutually non-overlapping objects then naive use of this approach can lead to unsatisfactory performance because the number of linear constraints will be quadratic in the number of objects since there is a non-overlap constraint between each pair of objects. Our third contribution is to propose a region-based approximation to handle non-overlap of many bodies. The idea is based upon grid and hierarchical spatial data-structures used for many geometric tasks such as detecting overlap between objects [15]. Instead of adding a constraint between each pair of objects, we divide space into regions, and use non-overlap constraints only between objects that overlap a common region but add “guards” to inform us when the object tries to enter a new region, i.e. add constraints requiring the object not to enter a new region and watch for when any of these constraints become active. Our fourth contribution is an empirical evaluation of our approach and comparison with the only comparable approach that we know of, a disjunctive approach to handling non-overlap constraints [12]. All software discussed in this paper, including the examples and timing code, is available for public download under terms of the GNU General Public License.

Starting with Sutherland [17], 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. [16, 18]); linear arithmetic solver based (e.g. [3]); geometric solver-based (e.g. [5, 9]); and general non-linear optimisation methods such as Newton-Raphson iteration (e.g. [14, 7]). However, none of these techniques support the kind of complex multi-way constraints, such as non-overlap, considered here.

Non-overlap constraints have been considered by Baraff [1] and Harada, Witkin, and Baraff [6], who use a specialised force based approach, modelling the non-overlap constraint between objects by a repulsion between them if they touch. Our approach is technically quite different because it relies on incremental linear constraint solving. It is also more powerful because it can be applied to more than just non-overlap constraints and it handles

non-overlap constraints in conjunction with any sort of linear constraint.

Hosobe [8] describes a general purpose constraint solving architecture that handles non-overlap constraints and other non-linear constraints. The system uses variable elimination to handle linear equalities and a combination of non-linear optimisation and genetic algorithms to handle the other constraints. Our approach addresses the same issue but again is technically quite different.

The most closely related work is our earlier paper on solving disjunctive constraints for interactive graphical applications [12]. Like the present paper, this was motivated by the need to handle non-overlap constraints. There the idea was to model non-overlap constraints by a disjunction of linear constraints. In our context this is essentially the same as using each of the disjuncts as a linear approximation to the constraint. Two key requirements of this approach are that the disjunction is finite and fixed and that the disjuncts are not disjoint since only moves between disjuncts which share the current solution are allowed. The present paper substantially generalises this approach and relaxes these two requirements. Thus, as illustrated before, the approach presented here can handle non-overlapping circles while our previous approach cannot since there is no fixed finite disjunction of linear constraints which capture non-overlap. More generally, the present approach can handle containment and non-overlap constraints for objects with smooth boundaries and table-layout which the previous approach could not. Other advantages of dynamic linear approximation is that it naturally allows an efficient region based approximation approach to non-overlap and it allows approximations which are not conservative. Finally, our empirical evaluation shows that the dynamic linear approximation approach is considerably faster than the simplex-based disjunction solving algorithm given in [12].

It is also worth clarifying that although the hierarchical approach to non-overlap is based on spatial data structures used in geometric applications, the approach presented here is considerably more powerful than standard geometric approaches used in, for instance game programs, to detect when two objects collide (usually with significant performance limitations on the number of moving objects), see e.g. [10]. We allow objects' relative position and size to be specified using arbitrary linear constraints. Thus objects can be allowed to shrink or move if they are collided with and may be linked to other objects. Standard approaches to collision detection do not allow this flexibility.

## DYNAMIC LINEAR APPROXIMATION

We are interested in rapidly (re)-solving systems of constraints to support direct manipulation in interactive graphical applications. The underlying context is that we have graphical objects displayed on the screen whose geometric attributes are 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 $\theta$ 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]. 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. Using this idea it is simple to formalise the constraint-solving required during direct manipulation. The constraint solver must repeatedly resolve a system of form $C_{layout} \wedge C_{stay} \wedge C_{edit}$. where only the *edit constraints*, $C_{edit}$, change during direct manipulation. Layout relationships between the graphic elements (e.g. alignment, containment, left-of, ...) are captured in $C_{layout}$ which is a conjunction of constraints (with associated strengths). Both the edit constraints and the *stay constraints*, $C_{stay}$, are conjunctions of desired equalities of form $U = b$. These, respectively, reflect our desire to give the variable $U$ the new value $b$ or for it remain as close as possible to its current value $b$.

We can now describe our generic algorithm dla_solve for supporting constraint solving during direct manipulation with dynamic linear approximation of complex constraints. It is given in Figure 2. It is designed to support rapid resolving during direct manipulation and it is intended that it be repeatedly called with different parameter values in the inner solver's goal function (e.g. changing the preferred value for each of a pair of variables to correspond to the mouse pointer's new coordinates).

The key idea is that for each complex constraint $D$ we have a current linear approximation composed of the *approximation proper* $C$ and a *guard* $G$ specifying the region in which $C$ is a good approximation to $D$. $G$ must be a conjunction of 'required'-strength linear inequalities, while $C$ can be a conjunction of arbitrary linear equalities and inequalities of any strengths. The approximation also has an arbitrary *test*, $T$, which checks the current solution $\theta$. If this test fails, this indicates that it may be possible to simplify or improve the current linear approximation.

```
dla_solve(L, D, G, T, C)
let D be of form D_1 ∧ ··· ∧ D_n
let G be of form G_1 ∧ ··· ∧ G_n
let T be of form T_1 ∧ ··· ∧ T_n
let C be of form C_1 ∧ ··· ∧ C_n
repeat
  L_approx := L ∧ (G_1 ∧ C_1) ∧ ··· ∧ (G_n ∧ C_n)
  θ := lsolv(L_approx)
  finished := true
  for i := 1, ..., n do
    let G_i be of form g^1 ∧ ··· ∧ g^m
    if not T_i(θ) or active(g^j) for some j = 1, ..., m then
      ⟨G', T', C'⟩ := lin_approx(D_i, G_i, C_i, θ)
      if G_i ≠ G' or C_i ≠ C' then
        G_i := G'
        T_i := T'
        C_i := C'
        finished := false
      endif
    endif
  endfor
until finished
return θ
```

Figure 2: Generic algorithm for constraint solving using dynamic linear approximation.

The algorithm takes a conjunction of linear arithmetic constraints $L$ which model linear layout relationships such as alignment or left-of and contains the stay and edit constraints. More interestingly, it takes a conjunction of complex constraints $D$ which has form $D_1 \wedge \cdots \wedge D_n$ along with the current guard $G_i$, approximation proper $C_i$, and test $T_i$ for each $D_i$. These complex constraints capture non-linear layout constraints such as non-overlap, or containment in non-convex shapes etc. These may be required or not.

The algorithm is parametric in the choice of an underlying linear constraint solver *lsolv* which takes a conjunction of linear arithmetic constraints, including stay and edit constraints, and returns their solution $\theta$.[1] Another parameter of the algorithm is the function *lin_approx*. This takes a complex constraint $D$, the current linear approximation, $G$ and $C$, to $D$, and the current solution $\theta$ and returns $\langle G', T', C' \rangle$, a possibly different linear approximation. The algorithm relies on the function *active* to determine if it might be worthwhile to move out of the region of the current approximation.

Our implementation allows the use of different definitions for active. The first is the standard definition of what it means for an inequality, $s \leq t$, to be active, namely it is active if $\theta(s) = \theta(t)$ [4] where we use the notation $\theta(e)$ to denote the result of evaluating expression $e$ using the assignment to variables given in $\theta$. Thus, in this case one of the guards will be active iff the current solution $\theta$ is on the boundary of the approximation region. The second definition we have used is to require that the inequality is *strongly active* meaning that it is active in the above sense and that the Lagrangian multiplier associated with the constraint in the Kuhn-Tucker system is strictly positive [4].[2] This basically means that the constraint is blocking movement in the direction of a more optimal solution, and so if it is removed we would hope to find a better solution.

The algorithm is extremely simple. It simply uses *lsolv* to compute the current solution $\theta$ using the current linear approximation to each complex constraint. Then it iterates through each approximation to see if there are any guard constraints which are active or any tests that no longer hold. If there are, then *lin_approx* is called to update the linear approximation used. The process continues until *lin_approx* does not change the linear approximation for any of the complex constraints, in which case the algorithm terminates and returns the solution to the current linear approximation.

## EXAMPLES OF LINEAR APPROXIMATION

Clearly, usefulness of the generic algorithm depends on us being able to define *lin_approx* for useful classes of complex constraints and being able to argue that the definitions are correct and will not lead to non-termination. Here we give five representative examples based on non-overlap and containment. We will consider termination and correctness in the next section.

### Non-overlap between circles

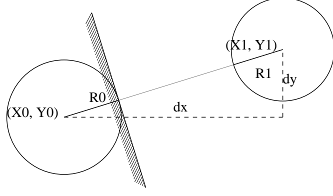
First we look at our motivating example from the Introduction: non-overlap of two circles. The definition of *lin_approx* for this constraint is given and illustrated in Figure 3. Note that the circle centers and radii are variables, not constants, so may have other constraints placed on them. We compute the distance in the $x$ direction, $dx$, in the $y$ direction, $dy$ and the Euclidean distance $d$ between the current circle centers. Using these we construct the new guard constraint

$$(dx/d) \times (X_1 - X_0) + (dy/d) \times (Y_1 - Y_0) \geq R_1 + R_2.$$

This constrains circle $C_1$ to be in the half-plane which is to the right of the tangent of the circle $C_0$ at the point on the circle closest to the center of $C_1$. Note that this is a linear constraint in the variables $X_1, X_0, Y_1, Y_0, R_1$ and $R_2$. This is the largest half-plane that we could choose in the sense that it maximises the distance from the current position of $C_2$ to any point not in the half

[1] For simplicity we assume that the required constraints are satisfiable.

[2] One thing to be careful of when evaluating strong activeness is that the chosen linear constraints should be linearly independent.




lin_approx($D$,$G$,$C$,$\theta$)
let $D$ be the constraint circles $C_0$ and $C_1$ do not overlap
let each $C_i$ have center $(X_i, Y_i)$ and radius $R_i$
$dx := \theta(X_1) - \theta(X_0)$
$dy := \theta(Y_1) - \theta(Y_0)$
$d := \sqrt{\Delta X^2 + \Delta Y^2}$
$G := (dx/d) \times (X_1 - X_0) + (dy/d) \times (Y_1 - Y_0) \geq R_1 + R_2$
**return** $\langle G, (\lambda\theta.true), true\rangle$

Figure 3: Definition of $lin_approx$ for non-overlapping circles.

plane, with the added requirement that the half-plane excludes the whole of $C_1$. It can also be understood as the linearisation of the non-overlap constraint at the tangent point.[3] In this case this single constraint defines the region of the approximation and within this region the non-overlap constraint always holds so the approximation proper is $true$. Adequacy of the linear approximation does not depend on the current solution so the test is simply $(\lambda\theta.true)$, a function that always returns $true$.

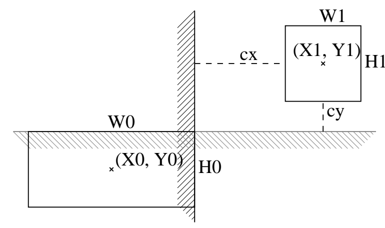
### Non-overlap between boxes

Non-overlap between boxes (i.e. axis-parallel rectangles) is easy to approximate, although it is difficult in general to guarantee that a local minimum will be found. An algorithm is given and illustrated in Figure 4. Essentially, we approximate non-overlap of $R_0$ and $R_1$ by $R_0$ is left-of $R_1$, $R_1$ is left-of $R_0$, $R_0$ is below $R_1$ or $R_1$ is below $R_0$. The code is slightly tricky so as to reduce the number of cases. We first compute the quadrant, represented by the sign pair $(qx, qy)$, that $R_1$ is in relative to $R_0$ and then determine the distance between them in the $x$ direction $(cx)$, and the $y$ direction $(cy)$. Based on their relative magnitude we either choose to approximate in the $x$ or the $y$ direction.

### Containment within a non-convex polygon

Another important and interesting class of geometric constraints is containment. Linear constraints naturally model containment in a fixed convex polygon, but cannot model containment in non-convex polygons or regions with smooth boundaries such as circles.

[3] The linearisation of $f(\vec{x}) \geq 0$ is simply the first-order Taylor expansion around the current solution $f(\theta)+(\nabla f)^T \cdot (\vec{x}-\theta) \geq 0$ [4].




lin_approx($D$,$G$,$C$,$\theta$)
Let $D$ be the constraint that boxes $R_0$ and $R_1$ do not overlap.
Let each $R_i$ have center $(X_i, Y_i)$, width $W_i$ and height $H_i$.
By assumption the heights and widths are constrained to be positive.
$q_x := sgn(\theta(X_1) - \theta(X_0))$
$q_y := sgn(\theta(Y_1) - \theta(Y_0))$
$c_x := q_x(\theta(X_1) - \theta(X_0)) - 0.5(\theta(W_0) + \theta(W_1))$
$c_y := q_y(\theta(Y_1) - \theta(Y_0)) - 0.5(\theta(H_0) + \theta(H_1))$

**if** $(c_x \leq c_y)$ **then**
  $Guard := q_y \times (Y_1 - Y_0) - 0.5(H_0 + H_1) \geq 0$
**else**
  $Guard := q_x \times (X_1 - X_0) - 0.5(W_0 + W_1) \geq 0$
**endif**
**return** $\langle Guard, (\lambda\theta.true), true\rangle$

Figure 4: Definition of $lin_approx$ for non-overlapping boxes.

Dynamic linear approximation is well-suited to modelling containment in a non-convex polygon. Consider containment of a point. The idea is that the local approximation is simply containment in a maximal convex polygon which is contained in the non-convex polygon and which contains the point. The choice of convex polygon approximation is changed whenever the point reaches a point on the perimeter of the convex polygon which is not a point on the perimeter of the non-convex polygon. Note that faces of the convex polygon are guards whenever they are partly contained within the interior of the surrounding non-convex polygon.

This is illustrated in Figure 5. In (a) we show the “C”-shaped non-convex polygon, the current location of the point (the “×”), the current convex approximation to the non-convex polygon, and the desired location of the point (the “∗”). Using this approximation the new current solution is on the boundary of the convex polygon approximation. This is shown in (b). Since a guard constraint is now active a new convex polygon approximation is computed. This is shown in (c). With this approximation the point can now take the desired value.

### Containment within a circle

Modelling containment within a region with smooth boundaries is more complex. Suppose that we want to model the constraint that a point is contained within a circle. One way is to use a similar idea to that used to

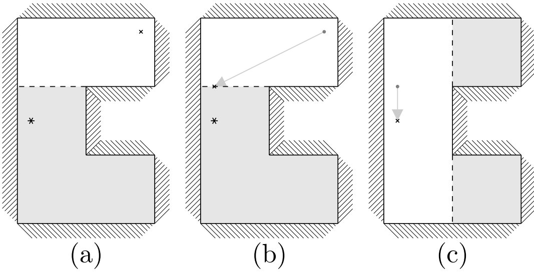



Figure 5: Constraining a point to lie inside a non-convex polygon.

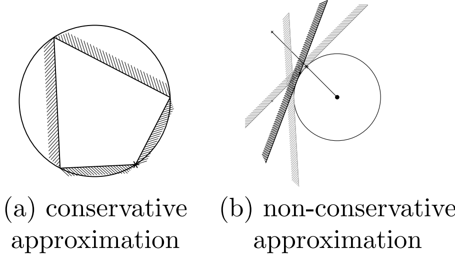



Figure 6: Constraining a point in a circle.

model containment in a non-convex polygon. The idea is to model containment within a circle by containment within a (convex) polygon whose vertices are on the circle. Clearly this is a safe approximation.

Figure 6 (a) illustrates the idea. We use a kite-shaped quadrilateral as the approximating polygon. Whenever the current solution is on the boundary of the kite we compute a new linear approximation by rotating the kite so that the thick base of the kite is centered around the current solution. Unfortunately, this approach does not work very well: if the current solution gets stuck in the vertex at the base of the kite it may not converge to the local optimum.

We have found that in practice a better approach is to model containment using a "non-conservative" approximation in which the approximating polygon is not fully contained in the circle. This idea is illustrated in Figure 6 (b). Essentially it works in the same way as the approximation for non-overlap of a circle and a point except that the inequality has the opposite direction. However, when computing the new approximation we also leave the linear constraint from the last approximation so at any time we have two constraints in the approximation. Since the approximation is not conservative we have a test to check that the approximation is correct at the current solution. I.e., the test is that the point is in fact within the circle. This is a good example of why guards are not enough and why we may need non-linear tests to check that a linear approximation is still safe.

## CORRECTNESS AND TERMINATION

Clearly, the definition of *lin_approx* is crucial to correctness and termination of our generic algorithm. But since its definition depends on the complex constraint being approximated, it is difficult to give general requirements. We cannot expect the algorithm to always find the globally best solution. This is because solving complex constraints is usually NP-hard (for example non-overlap constraints with even simple shapes) and our primary concern is to ensure sufficient speed for direct manipulation. The best that we can hope to prove is that: (1) the algorithm terminates, (2) it is correct in the limited sense that it returns a solution satisfying the desired constraints, and (3) it returns a solution which is a local optimum taking into account the desired constraints such as the stay and edit constraints.

One way of ensuring that the algorithm is correct in the above sense is to require that for a required complex constraint $D$ that each linear approximation is *conservative* in the sense that $G \rightarrow (D \leftrightarrow C)$ where $G$ and $C$ are the linear approximation's guard and approximation. This states that within the solution space of the guard, $D$ and $C$ have exactly the same solutions. Thus, any solution of the linear approximation is a solution of the complex constraint. All of the examples given in the previous section, except for the non-conservative approximation to point containment within a circle are conservative in this sense.

For non-conservative approximations correctness is ensured by requiring that *lin_approx* returns a different approximation whenever the current solution to the linear approximation does not satisfy the complex constraint. The non-conservative approximation to point containment within a circle is therefore correct since it satisfies this requirement.

A general schema for arguing termination is to prove the following: (a) the quality of the current solution $\theta$ never decreases after the next iteration, (b) there can only be a finite number of iterations in which the quality remains the same, and (c) we cannot improve solution quality indefinitely. Clearly together these imply termination. Ensuring (a) is relatively easy. For conservative approximations, a sufficient condition for ensuring that the quality of solution can never decrease is that if $lin_approx(D, G, C, \theta)$ returns $\langle G', T', C' \rangle$ then $\theta$ is a solution of $G'$, since this ensures that whatever solution is returned in the next approximation must be at least as good as $\theta$. This requirement also reflects our desire for dynamic linear approximation to behave in a smooth, continuous fashion. All of the conservative approximations given as examples satisfy this requirement. In the case of non-conservative linear approximations, we can require that the new approximation rules out any solu-

tion that less satisfies (by some convenient metric) the constraint than the current solution $\theta$. (This is equivalent to considering constraint satisfaction as part of the objective function, as is done when using an 'exact penalty function'.) Ensuring (b) may depend more on the exact combination of constraints. One simple way of ensuring it is to modify the generic algorithm to terminate if the solution quality does not improve after a few iterations. Condition (c) holds in a weak sense assuming limited precision floating point numbers for any reasonable system of constraints. However, another sufficient condition for (c) is that each complex constraint has only a finite number of linear approximations.

Ensuring that a solution $\theta$ returned from the algorithm is a local optimum is difficult. By assumption, the solution returned by *lsolv* is at least a locally optimal solution to the linear approximation of the problem. It will be a locally best solution to the original problem if the local approximation at the point $\theta$ is accurate enough to ensure that $\theta$ satisfies the second order sufficient conditions (for optimality) [4] for the original problem. This mostly holds in our examples. However, in the case of containment and non-overlap of polygons such as boxes, this breaks down in the case that the solution is on the vertex of the polygon.

In practice in interactive graphical applications, if the user is unhappy with the solution found then they can usually perform additional manipulation to get to the solution they want.

## EFFICIENT MUTUAL NON-OVERLAP OF MANY OBJECTS

If we wish to model $n$ mutually non-overlapping objects then the obvious approach is to add a non-overlap constraint between each pair of objects, giving rise to $\frac{n(n-1)}{2}$ overlap constraints and hence a quadratic number of linear constraints. When $n$ is large this naive approach can lead to unsatisfactory performance.

A similar problem arises when detecting collision of a single moving body with a large number of fixed objects (a problem arising in gaming programs). The obvious solution is to perform a linear number of comparisons, one with each object. However, use of spatial data structures such as grids and hierarchical data structures such as BSP-trees allow collision detection to be performed much faster, in the best case in virtually constant time [10]. The idea is that space is partitioned into regions, and one only needs to check for possible collision between the moving object and those fixed objects in the same region.

We can use a similar idea to greatly reduce the number of linear constraints required to model mutual non-overlap. We first partition space into a number of rectangular regions. We then compute for each object $o$ the minimal set of these rectangular regions whose union forms a convex region (actually a rectangle) and which contains the object. We call this region the *bounding region* of $o$. We add a non-overlap constraint between any pair of objects whose bounding regions intersect and guard constraints to ensure that an object remains within its bounding region. This is a safe approximation since if an object remains inside its bounding region it cannot overlap with objects that do not intersect this region.

The important point is that if we assume that there is an upper bound on the number of objects whose bounding regions overlap, say $k$, then we have $n$ containment constraints and $n(k-1)$ overlap constraints which is a considerable improvement over $n^2$ assuming that $n \gg k$. For instance, consider the boxes shown in Figure 7(a). Since there are 13 boxes naive application of the non-overlap solver for boxes will give rise to $(13 \times 12)/2 = 78$ linear constraints. Now look at the approximation shown in Figure 7(b) where we use a grid with 6 cells. Each box is contained to stay within its bounding region, in this case the rectangle of grid cells intersecting the object, which gives $13 \times 4 = 52$ linear constraints, plus the 14 linear constraints for non-overlap between boxes whose bounding regions intersect (these are shown as bi-directional arcs between the two boxes involved). Thus using the partition approach the number of constraints is substantially reduced and as the number of objects increases so does the benefit of this approximation.

Constraint solving during direct manipulation works as follows with the partition approximation. When an object is moved its bounding region may change. Change of the bounding region is triggered by the object touching a boundary of the region and so the associated inequality becoming active. Whenever the bounding region changes we must add non-overlap constraints for the new objects whose bounding region now intersects the new bounding region. Tests are used to determine when the moving object's bounding region shrinks, potentially allowing non-overlap constraints to be removed.[4] Consider Figure 7(a) again and imagine that we wish to move the box labelled E down to the right of box L. As shown in Figure 7 (c), box E can be moved until it hits the lower boundary of its bounding region. This is a guard constraint. Once this becomes active a new approximation is computed in which box E's new bounding region are the two rightmost cells. Since this intersects the bounding region of boxes L and M, two additional non-overlap constraints must be added. This is shown in Figure 7 (d). The approximation sets up a test that

[4]These could be modelled using guards but it is more efficient to use a test since doing so does not affect safety of the approximation and tests do not add linear constraints to the solver.

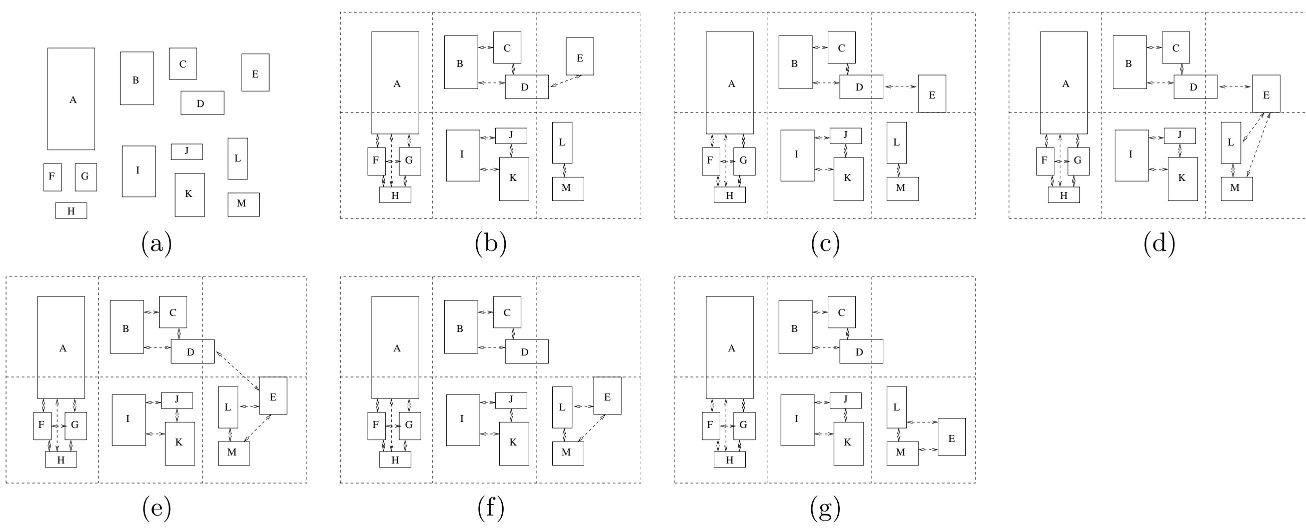



Figure 7: Approximating mutual non-overlap.

checks that the box is still straddling the two grid cells. Once the box moves completely within the bottom grid cell (as shown in Figure 7 (e)) the test fails and a new approximation is computed. This time the bounding region of the box becomes the bottom grid cell and the non-overlap constraint between box E and box D can be removed. This is shown in Figure 7 (f). Finally the box is moved to the desired location as shown in Figure 7 (g).

This is a good example of how, when we have the choice between a simple linear approximation with only a few linear linear constraints and a more complex, hence expensive, approximation which is more general, we can control the approximation so as to use the simple linear approximation when possible and only use the more complex one when required. Guards control the change from the simple approximation to the more complex but general approximation while tests control the change in the other direction.

## EMPIRICAL EVALUATION

In this section we provide a preliminary empirical evaluation of dla_solve. Our implementation uses the C++ implementation of the Cassowary Algorithm in the QOCA toolkit [11] as the underlying linear constraint solver and the simple definition of active: namely a linear inequality is active whenever its slack variable is zero. All times are in milliseconds measured on a 400MHz Pentium II based computer using gcc 2.95.4 on linux 2.2.9. (Granularity of maximum re-solve times is 10ms.)

Our first two experiments compare the speed of dla_solve with the only comparable solver, the the Disjunctive Simplex Solver (disj_simplex_solve) given in [12]. We use the benchmarks used in [12] for the comparison. The first experiment also allows us to compare the overhead of dla_solve with the underlying Cassowary Algorithm.

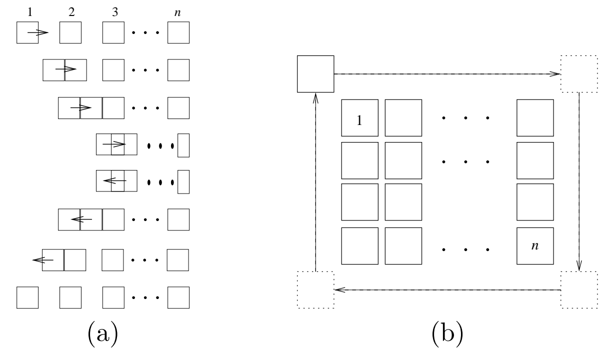



Figure 8: Experiments to compare dla_solve with disj_simplex_solve. (Figures are taken from [12])

The first experiment is illustrated in Figure 8(a). It 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 while the others are free to move, but have a stay constraint tending to keep them at their current location. For the dla_solve and disj_simplex_solve version of the problem we add a non-overlap constraint between each box and every other box. 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 linear approximation chosen in the dla_solve version and the active disjunct 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 shown in Table 1 gives the number $n$ of boxes, for each version the num-

| $n$ | Cassowary Cons | Cassowary AveR | Cassowary MaxR | disj_simplex_solve Cons | disj_simplex_solve AveR | disj_simplex_solve MaxR | dla_solve Cons | dla_solve AveR | dla_solve MaxR |
|---|---|---|---|---|---|---|---|---|---|
| 20 | 190 | 2.95 | 10 | 760 | 16.54 | 30 | 190 | 3.84 | 20 |
| 40 | 780 | 11.96 | 30 | 3120 | 81.96 | 120 | 780 | 15.23 | 40 |
| 60 | 1770 | 29.20 | 60 | 7080 | 200.13 | 290 | 1770 | 35.14 | 80 |
| 80 | 3160 | 56.32 | 100 | 12640 | 360.31 | 540 | 3160 | 64.99 | 120 |

Table 1: Relative overhead of disj_simplex_solve and dla_solve.

| $n$ | disj_simplex_solve AveR | disj_simplex_solve MaxR | dla_solve–box AveR | dla_solve–box MaxR | dla_solve–circle AveR | dla_solve–circle MaxR |
|---|---|---|---|---|---|---|
| 200 | 56.54 | 1010 | 15.33 | 70 | 20.49 | 80 |
| 400 | 197.54 | 2320 | 49.85 | 210 | 67.04 | 180 |
| 600 | 428.53 | 3810 | 107.16 | 440 | 139.58 | 320 |
| 800 | 723.41 | 5190 | 180.14 | 700 | 213.04 | 420 |
| 1000 | 1065.98 | 6640 | 248.59 | 990 | 294.39 | 600 |
| 1200 | 1453.67 | 8110 | 322.85 | 1440 | 385.66 | 740 |

Table 2: Comparison of disj_simplex_solve and dla_solve.

ber of linear constraints (Cons) in the solver, the average time (AveR), and maximum time (MaxR) to resolve during the direct manipulation (in milliseconds). The results show that disj_simplex_solve has considerably greater overhead than dla_solve. Indeed dla_solve is only slightly slower than Cassowary, which is very pleasing. The overhead for disj_simplex_solve probably occurs because it keeps non-active disjuncts in the solved form as well as the active disjunct corresponding to the current linear approximation leading to more constraints in the simplex tableau.

Our second experiment gives a comparison between the speed of dla_solve and disj_simplex_solve when linear approximations change and disjunct swapping takes place. Figure 8(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 2 gives the number $n$ of boxes, and for each solver dla_solve and disj_simplex_solve–box the average and maximum time for each resolve in milliseconds. Again we see that dynamic linear approximation is considerably faster both on average and in the maximum than disj_simplex_solve.

Our third experiment is similar to the second except that instead of boxes we use circles and so use the non-overlapping circle algorithm instead of a non-overlapping box algorithm. It is not possible to compare the speed with disj_simplex_solve since this cannot handle non-overlap between circles. The average and maximum time for each resolve using the same values of $n$ are also given in Table 2 under the heading dla_solve–circle. Interestingly, the performance is very similar to that for non-overlapping boxes.

| $n$ | simple AveR | simple MaxR | partition AveR | partition MaxR |
|---|---|---|---|---|
| 21 | 10.86ms | 680ms | 4.31ms | 100ms |
| 41 | 108.27ms | 11240ms | 9.23ms | 360ms |
| 61 | 393.62ms | 55000ms | 16.38ms | 760ms |
| 81 | 905.00ms | 163220ms | 29.41ms | 1790ms |

Table 3: Comparison of simple $O(n^2)$ and partition approximation to mutual non-overlap.

Our final experiment compares the partition approximation to mutual non-overlap of $n$ bodies with the simple $n^2$ approach. We do this using a slight modification of Experiment 2. Again we move a single box around a rectangle of $n$ boxes with the constraint that this box does not overlap any of the other boxes. This time, however, we also add the constraint that no boxes in the rectangle can overlap. In the simple approach we add $O(n^2)$ non-overlap constraints, one for each pair of boxes. In the partition approximation approach we use $n/4$ square-shaped partitions each containing 4 squares. The underlying solver used for handling non-overlap is disj_simplex_solve rather than dla_solve. This does not affect the validity of the comparison and reflects that we were developing this implementation in parallel with that of dla_solve. The results are shown in Table 3. As one would expect for even moderate sized $n$, the partition approach is considerably faster; the naive approach

is only suitable for small $n$.

As a final argument for the practicality of dynamic linear approximation, we have implemented a simple graphics editing toy which provides circles and boxes and allows the user to add non-overlap constraints between these, containment, alignment and equal distribution. Non-overlap between circles and boxes uses the algorithms detailed here. Containment of a box in a circle is simply modelled by containment of the box's corners in the circle using the non-conservative approximation. Other constraints are modelled directly by linear constraints. Despite its immaturity, the performance of the editing toy is impressive. It provides direct manipulation in real-time with dozens of objects and hundreds of constraints.

## CONCLUSIONS

We have described how dynamic approximation by linear constraints provides a simple yet powerful approach to solving complex geometric constraints such as non-overlap and containment. Our generic algorithm is designed to support constraint solving arising in direct manipulation. Empirical evaluation of the approach is very encouraging and suggests that dynamic linear approximation is efficient enough for practical applications.

The approach is very general but may appear somewhat ad hoc (we originally coined the name "hactive set" for the approach) since it requires a specialised approximation algorithm to be programmed for each complex constraint we are interested in. However, this allows the programmer to take advantage of application specific heuristics, much like the way in which the behaviour of complex constraints is explicitly programmed in propagation-based constraint solvers [13]. Given that one is dealing with NP-hard problems this "constraint programming" approach may well be the only practical approach.

## REFERENCES


1. D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94 Conference Proceedings*, pages 23–32, 1994. ACM.
2. A. Borning, B. Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.
3. A. Borning, K. Marriott, P. 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*, New York, October 1997. ACM.
4. R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 1987.
5. I. Fudos. *Geometric Constraint Solving*. PhD thesis, Purdue University, Dept. of Computer Sciences, 1995.
6. M. Harada, A. Witkin, and D. Baraff. Interactive physically-based manipulation of discrete/continuous models. In *SIGGRAPH '95 Conference Proceedings*, pages 199–208, Los Angeles, August 1995. ACM.
7. A. Heydon and G. Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, DEC Systems Research Center, Palo Alto, CA, 1994.
8. H. Hosobe. A modular geometric constraint solver for user interface applications. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, New York, November 2001. ACM.
9. G. Kramer. A geometric constraint engine. *Artificial Intelligence*, 58(1–3):327–360, December 1992.
10. M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *IMA Conference on Mathematics of Surfaces*, 1998.
11. 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.
12. K. Marriott, P. Moulder, P. Stuckey, and A. Borning. Solving disjunctive constraints for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming (CP01)*, 2001.
13. K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.
14. G. Nelson. Juno, a constraint-based graphics system. In *SIGGRAPH '85 Conference Proceedings*, pages 235–243, San Francisco, July 1985. ACM.
15. H. Samet. *Spatial Data Structures: Quadtree, Octrees and Other Hierarchical Methods*. Addison Wesley, 1989.
16. M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.
17. I. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Dept. of Electrical Engineering, MIT, January 1963.
18. B. 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.