An Incremental Planning Algorithm for Ordering Equations in a Multilinear System of Constraints

Bradley T. Vander Zanden*

88-910
April 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This research was supported by a National Science Foundation grant DCR 86-02663.
AN INCREMENTAL PLANNING ALGORITHM FOR ORDERING EQUATIONS IN A MULTILINEAR SYSTEM OF CONSTRAINTS

Bradley T. Vander Zanden
Cornell University
Ithaca, NY 14853

Abstract

Constraint equations are increasingly being used in interactive applications such as graphics, logical programming, and simulation that demand immediate feedback. To handle the performance requirements imposed by such systems constraint evaluators must use incremental satisfaction techniques. In this paper we apply these techniques to noncircular, multilinear systems of equations. The constraint satisfaction process is divided into two phases—a planning phase that imposes a topological order on the equations and an execution phase that evaluates the equations. A planning algorithm is presented that incrementally updates this order each time the constraint system changes. This technique achieves significant performance improvements in large constraint systems since modifications generally perturb only a small portion of the topological order.

1 Introduction

Constraint equations are finding a growing use in many interactive applications, particularly graphics, logical programming, and simulation. The reason? Constraints provide a viable and flexible vehicle for representing relationships in diverse areas such as document preparation [Van Wyk 82], graphical layout [Gosling 83] [Nelson 85] [Leler 88], the simulation of physical systems [Borning 81,86], and animation [Duisberg 86]. However, if constraints are to afford a practical and workable option, they must be solved rapidly enough to give users a sense of immediate feedback. That is, when users modify the constraints in some manner, the results should be available almost instantaneously. Since it is too time-consuming to resolve the entire set of constraints in response to each operation, some other approach is required. An incremental recomputation of the constraints provides such an approach: it takes the previous constraint solution and updates it.

In this paper we restrict our attention to multilinear, noncircular sets of constraints and explore techniques for incrementally reevaluating such constraints. In a companion paper we present heuristics for minimizing the number of reevaluated constraints [Vander Zanden 88a]. A function is multilinear if its value changes by a linear amount when any of its variables changes by a linear amount. This definition may be formally written as $f(x_1, ..., ax_j + b, ..., x_n) = af(x_1, ..., x_j, ..., x_n) + f(x_1, ..., b, ..., x_n)$ [Edwards 73]. A set of equations is noncircular if at every step of the evaluation process, it is possible to find an

---

1 This research was supported by a National Science Foundation grant DCR 86-02663.
equation that is solvable given the results of previously evaluated equations. That is, a simultaneous equation solver is not needed.

Multilinear systems of constraints are useful in defining the graphical layout of an application. The objects used in many two dimensional graphical systems have relatively simply geometric shapes—rectangular bitmaps, circles, collections of line segments, and rectangular boxes of text. The graphical relationships between these objects can generally be modeled by multi-linear sets of constraints:

1. centering, left or right justifying objects, especially text;
2. constraining objects to stay within a rectangular area;
3. computing the space required by objects (e.g., the space occupied by a component is the sum of the space occupied by its subcomponents);
4. constraints on the dimensions of objects such as their heights or widths; and
5. constraints on the angles between line segments (an arrow or resistor for example).

Published algorithms that solve these constraint systems have taken two separate approaches to the problem. One approach separates the solution process into a planning phase that locates and topologically orders a set of unsatisfied constraints and an evaluation phase that reevaluates these constraints [Sutherland 63][Borning 81]. The problem with this approach is that the planning phase computes the evaluation sequence from scratch after the constraint system changes. However changes to the constraint system generally cause only a local perturbation to the evaluation sequence and thus this approach may perform unnecessary work. The second approach avoids the planning phase and immediately begins reevaluating constraints [Gosling 83]. This technique tests the values of variables to determine if they have changed and quiesses when there are no more unsatisfied constraints. Unfortunately, this algorithm may evaluate an exponential number of equations.

This paper presents an algorithm that is based on the first approach. The performance of the planning phase is improved by introducing an algorithm that incrementally reorders the equations in a noncircular, multilinear constraint system. In large constraint systems this technique can significantly decrease planning time since modifications generally perturb only a small portion of the topological order.

Because the planning algorithm does not explicitly assign order numbers to the equations, we present three evaluation algorithms that have varying degrees of success in observing the topological order that the planning algorithm establishes. The most successful scheme in this regard is based on approximate topological ordering [Hoover 86, 87]. In this algorithm, the planning phase incrementally updates the topological order and assigns order numbers that represent its best guess as to where the equation is located in this sequence. The evaluation algorithm generally evaluates a polynomial number of equations, although in the worst case it may deviate from this order and evaluate an exponential number of equations. In a companion paper [Vander Zanden 88a], we present a planning algorithm that explicitly assigns order numbers to equations and thus assures
that the evaluator solves only a polynomial number of equations.

The rest of this paper is organized as follows. In section 2 we discuss related work. In section 3 we present the basic, nonincremental algorithm for evaluating a noncircular, multilinear system of constraints. In section 4 we show how the planning phase of this algorithm can be made incremental. In section 5 we present several algorithms that evaluate the sequence of equations generated by the planning algorithm and in section 6 we provide some experimental comparisons between these algorithms. Finally, in section 7 we present our conclusions.

2 Related Work

General purpose nonlinear problem solvers such as Newton’s method or the method of secants can be used to solve noncircular, multi-linear systems of constraints [Dennis 83]. However, these methods do not exploit the noncircular aspect of the constraints and thus are not the most efficient algorithms available. These algorithms are also fairly inflexible since they cannot symbolically manipulate the constraints.

Emanuel Derman and Christopher Van Wyk [Derman 84] have presented an algorithm whose running time is quadratic in the number of constraint equations for circular, multi-linear systems of constraints and which is able to do symbolic manipulation. However, this algorithm cannot be readily adapted to efficiently satisfy a noncircular set of constraints.

The first reference that the author is aware of that exploits the noncircular structure of a constraint system is Ivan Sutherland’s SketchPad system [Sutherland 63]. This system divides the satisfaction process into a planning phase and an execution phase. The planning phase uses a technique known as propagation of degrees of freedom to topologically order the equations. The execution phase then uses this order to solve the equations. This algorithm is linear in the number of equations in the system but has the drawback that each time the constraints are changed, the evaluation sequence is recomputed from scratch.

Alan Borning’s ThingLab improved this algorithm by sophisticated the planning process [Borning 81]. Rather than relying on a global analysis that is independent of the change to the constraint network, ThingLab’s algorithm uses a local analysis that starts at the point of change. It marks all constraints that depend directly or indirectly on the changed constraint and then recomputes the evaluation sequence for this set of constraints. If the change affects an area of the constraint network that is considerably smaller than the size of the entire network, then the performance of the planning phase improves considerably. The drawback to this approach is that the region reachable from the initial point of inconsistency is often proportional to the size of the constraint network.

James Gosling noted that planning depends on the structure of the constraint network and not the updated values of the variables [Gosling 83]. Consequently, it may examine more equations than necessary if the updated values of some variables are unchanged.
Thus Gosling's Magritte system avoided the planning step and proceeded immediately to the evaluation phase. Magritte seeks to minimize the number of constraints that have to be reevaluated. Each time a constraint changes, its algorithm initiates a breadth first search that starts at the affected constraints. The algorithm successively considers all valid sequences of constraints of length one, two, three, and so on until a sequence that produces an acceptable solution is found. This search ensures that the minimal sequence is always found but it may require exponential time to find the sequence. For the constraint networks handled by his system, Gosling found that only a small number of valid sequences existed and thus the performance of the algorithm was acceptably fast. However, it is doubtful that this performance would extend to large constraint systems.

3 A Nonincremental Planning Algorithm

In this section we will provide an overview of the propagation of degrees of freedom algorithm. This algorithm is nonincremental since it throws away the old evaluation sequence and completely reconstructs a new one from scratch. In the next section, we will discuss techniques that make this algorithm incremental. The number of equations examined by these techniques will be proportional to the number of equations affected by the changes to the constraint network.

In the ensuing discussion we will occasionally use hypergraphs to illustrate a point. A hypergraph is similar to an ordinary graph except that the edges of a hypergraph may contain one or more vertices. In the hypergraphs presented in this paper, the vertices represent variables and the edges represent equations. Each equation is represented by a single edge and the vertices in this edge correspond to the variables in the equation. Formally we can represent a constraint system as the hypergraph $H = \langle V, E \rangle$ where $V$ represents the set of variables contained in the constraint system and $E$ represents the equations of the constraint system. A sample hypergraph and the constraint system it represents is shown in figure 1.

![Hypergraph representation of a set of constraints that describe the graphical layout of a pair of rectangles.](image)

1. $r1.ne = r1.nw + r1 wd$
2. $r1.se = r1.ne + r1 ht$
3. $r1.se = r1.sw + r1 wd$
4. $r2.ne = r2.nw + r2 wd$
5. $r2.se = r2.ne + r2 ht$
6. $r2.se = r2.sw + r2 wd$
7. $r1.ht = r2.ht$
8. $r2.nw = r1.ne + r1 wd$
A planning algorithm based on propagation of degrees of freedom enumerates equations in reverse topological order. Initially it looks for an equation that contains a free variable, a variable that belongs to only one equation. Such an equation is said to have a degree of freedom since one of its variables can be modified without affecting the rest of the constraint system. Consequently these equations can be placed last in the topological order.

Once the planning algorithm finds such an equation, it inserts the equation and its free variable into the evaluation sequence and eliminates the equation from the constraint system. The algorithm then repeats this process on the reduced system, taking care to always place the chosen equation-variable pair at the head of the sequence. The algorithm acquired its name because it propagates degrees of freedom from the equations it eliminates to the equations that remain in the constraint system.

One of the possible sequences that could be produced by the planning algorithm for the constraint system shown in figure 1 is

\[(6, r2.sw), (8, r1.wd), (4, r2.ne), (5, r2.ht), (7, r1.ht), (2, r1.se), (3, r1.sw), (1, r1.nw)\]

For any given constraint system, there are often many valid sequences since at any step of the algorithm, there are normally several equations with at least one degree of freedom and thus the equation to be eliminated is chosen arbitrarily. For example, once the sequence \[(2, r1.se), (3, r1.sw), (1, r1.nw)\] has been chosen, equations 6, 7, and 8 all have at least one degree of freedom. This assertion can be verified by examining the hypergraph of the reduced system in figure 2. For the above sequence, the algorithm chose to eliminate equation 7 and chose the free variable \(r1.ht\) as the variable that the equation is solved for.

The planning algorithm is formalized in figure 3. The running time of the equation is linear in the number of equations, \(O(|E|)\) and quadratic in the number of variables, \(O(|V|^2)\).

![Figure 2: Hypergraph representation of the constraint system in figure 1 with equations 1, 2 and 3 and the variables r1.nw, r1.se, and r1.sw eliminated.](image-url)
propagate_degrees_of_freedom
let E = the set of equations in the constraint system
Seq = the evaluation sequence created by the planning phase
S = the set of equations with free variables
Seq = Ø
S = Ø

/* if an equation e contains a free variable, add it to S */

for each e ∈ E do
    if degrees_of_freedom(e) ≥ 1 then
        v = find_free_var(e)
        solved(e) = true
        S = S ∪ {(e,v)}

/* propagate degrees of freedom */

while S ≠ Ø do
    (e,v) = delete_element(S)
    Seq = Seq ∪ {(e,v)}
    for each w ∈ e, v ≠ w do
        numeqn(w) = numeqn(w) - 1
        if numeqn(w) = 1 then
            e' = find_free_eqn(w)
            if solved(e') = false then
                S = S ∪ {(e',w)}
                solved(e') = true

Figure 3: A nonincremental algorithm for topologically ordering equations.

4 An Incremental Planning Algorithm

The planning algorithm presented in the previous section recomputes the evaluation sequence from scratch each time the constraint system changes. Such a change could take a variety of forms—adding or deleting a variable from an equation, adding or deleting an equation, changing the value of one or more variables, or fixing the values of one or more variables. Quite often these changes will cause only a local perturbation in the evaluation sequence. In this section we will explore techniques that exploit this condition to reduce the amount of work performed in the planning phase.

We first describe the additional types of information we must maintain and then discuss how this information can be used to incrementally update the evaluation sequence. We only consider the deletion or insertion of an equation since changes that affect equations. For example, fixing or
changing the value of a variable is comparable to adding an equation that assigns a constant to the variable. Similarly, adding or deleting a variable from an equation is comparable to deleting and reinserting the equation. Of course, it is somewhat more efficient to write algorithms that directly handle variable modifications. However, these algorithms can be easily obtained from the algorithms that handle equation insertion and deletions and thus will not be further discussed in this paper.

In discussing equation deletions and insertions, we will first show how they can be handled separately and then simultaneously.

4.1 Data Structures

Our idea uses auxiliary data structures to maintain dependency information about the evaluation sequence. For each variable \( v \), we record the equation that it eliminates and the sequence of variable-equation pairs that fired (i.e., were eliminated) to free it. The data structure that records this sequence is called the fired structure, denoted fired\((v)\) and the data structure that records the eliminated equation is denoted eqn_elim\((v)\). For each equation we record the variable that it is solved for. This data structure is denoted var_elim\((e)\). Notice that the equations that belong to the fired structure of the variable that eliminates an equation must precede the equation in topological order. The initial algorithm that computes this dependency information is shown in figure 4. Notice that this algorithm is an expanded version of the planning algorithm presented in section 3.

In any noncircular, multilinear system of constraints the number of variables must be at least equal to the number of constraints and in interactive systems they generally exceed the number of constraints. For example, in figure 1, the constraint system has 8 equations and 12 variables. Thus several variables may eliminate the same equation. A variable is termed a designated variable if it is the variable designated to eliminate an equation (i.e., it belongs to the equation’s var_elim structure). Otherwise the variable is termed an excess variable. Interactive systems assign default values to the excess variables so that the equations can be initially solved.

The algorithm in figure 4 no longer explicitly records the evaluation sequence since it is implicitly recorded in the var_elim data structure.

Example: Suppose the algorithm given in figure 4 is applied to the constraint figure shown in figure 1. Figure 5 shows the values for the fired, eqn_elim, and var_elim data structures computed by this algorithm. The algorithm generates the sequence \([(6, r2.sw), (8, r1.wd), (4, r2.ne), (5, r2.ht), (7, r1.ht), (2, r1.se), (3, r1.sw), (1, r1.nw)]\). In computing these data structures, we have assumed that the set \( F \) is implemented as a stack so that the union operation is comparable to a push and the delete_element operation is comparable to a pop.

We can briefly trace the execution of the algorithm as follows. The initial search of the equations \( E \) finds three equations with at least one degree of freedom—equations 1, 3, and 6. The var_elim data structures for these three equations are given the variables \( r1.nw, r1.sw, \) and \( r2.sw \) respectively. Similarly the eqn_elim data structures for the variables \( r1.nw, r1.sw, \) and \( r2.sw \) are initialized to the equations 1, 3, and 6.
After some initial computation in which the sequence 
((3, r1.sw), (1, r1.nw)) is generated, the auxiliary data structures acquire the values shown in figure 6a. Figure 6b shows the state of these same structures after the algorithm has selected the pair (2, r1.se) from F and processed it. The algorithm adds this pair to the fired structures for the variables r1.ne and r1.ht. Both of these variables become free and add a degree of freedom to equations 8 and 7 respectively. These equations are noted in the variables eqn_elim data structure. Since equation 7 does not yet have an eliminating variable, r1.ht is designated as the eliminating variable and recorded in the eqn_elim data structure. Equation 8 already has an eliminating variable and thus r1.ne becomes an excess variable. Notice that when equation 8 is eventually eliminated, there is a conditional statement "if numeqn(w) - |fired(w)| > 1" that prevents this equation from being added to the fired structure of r1.ne. Finally, because equation 7 has obtained a degree of freedom, the pair (7, r1.ht) is added to F.

```plaintext
sequence_planning(F : set)
    let E = the set of equations in the constraint system
    F = the set of equations with free variables
    /* if an equation e initially contains a free variable, add it to F */
    for each e ∈ E do
        if degrees_of_freedom(e) ≥ 1 then
            v = find_free_var(e)
            solved(e) = true
            eqn_elim(v) = e
            var_elim(e) = v
            F = F ∪ [(e, v)]
    build_sequence(F)

/* planning phase */

build_sequence(F : set)

while F ≠ ∅ do
    (e, v) = delete_element(F)
    for each w ∈ e, v ≠ w do
        if (numeqn(w) - |fired(w)|) > 1
            and e ≠ member(fired(w)) then
            fired(w) = fired(w) ∪ [(e, v)]
            if (numeqn(w) - |fired(w)|) = 1 then
                e' = find_free_eqn(w)
                eqn_elim(e) = e'
                if solved(e') = false then
                    F = F ∪ [(e', w)]
                    solved(e') = true
                    var_elim(e') = w

Figure 4: Algorithm that computes the dependency information and the evaluation sequence the first time the constraint system is presented to the constraint solver.
<table>
<thead>
<tr>
<th>variable</th>
<th>fired structure</th>
<th>eqn_elim</th>
<th>equation</th>
<th>var_elim</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1.nw</td>
<td></td>
<td>1</td>
<td>1</td>
<td>r1.nw</td>
</tr>
<tr>
<td>r1.ne</td>
<td>(1,r1.nw), (2,r1.se)</td>
<td>8</td>
<td>2</td>
<td>r1.se</td>
</tr>
<tr>
<td>r1.se</td>
<td>(3,r1.sw)</td>
<td>2</td>
<td>3</td>
<td>r1.sw</td>
</tr>
<tr>
<td>r1.sw</td>
<td></td>
<td>3</td>
<td>4</td>
<td>r2.ne</td>
</tr>
<tr>
<td>r1.ht</td>
<td>(2,r1.se)</td>
<td>7</td>
<td>5</td>
<td>r2.ht</td>
</tr>
<tr>
<td>r1.wd</td>
<td>(1,r1.nw), (3,r1.sw)</td>
<td>8</td>
<td>6</td>
<td>r2.sw</td>
</tr>
<tr>
<td>r2.ht</td>
<td>(7,r1.ht)</td>
<td>5</td>
<td>7</td>
<td>r1.ht</td>
</tr>
<tr>
<td>r2.wd</td>
<td>(4,r2.ne)</td>
<td>6</td>
<td>8</td>
<td>r1.wd</td>
</tr>
<tr>
<td>r2.nw</td>
<td>(4,r2.ne)</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.ne</td>
<td>(5,r2.ht)</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.se</td>
<td>(5,r2.ht)</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.sw</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Values of the auxiliary data structures that the sequence_planning algorithm computes for the constraint system in figure 1.

### 4.2 Deleting an Equation

When an equation is deleted from the constraint system, the planning algorithm must incrementally recompute the dependency information in the auxiliary data structures. It does so in the following manner. First, notice that the variables that comprise an equation can be divided into two classes—those variables that include the equation in their fired structures and those variables that eliminate the equation. When an equation is deleted, these two classes are processed differently. A variable that includes the equation in its fired structure must have the equation removed from this structure. In contrast, a variable that eliminates the equation must be given a replacement equation to eliminate. Any equation in the variable's fired structure is a candidate and the planning algorithm arbitrarily removes one such equation and inserts it into the variable's eqn_elim structure. Deletions do not affect the evaluation sequence since they create excess variables but not designated variables. Thus the planning algorithm may terminate once the deleted equations have been processed. The portion of the planning algorithm for deleted equations is formalized in figure 7.
<table>
<thead>
<tr>
<th>variable</th>
<th>fired structure</th>
<th>eqn_elim</th>
<th>equation</th>
<th>var_elim</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1.nw</td>
<td>1</td>
<td>1</td>
<td>r1.ne</td>
<td></td>
</tr>
<tr>
<td>r1.ne</td>
<td>(1,r1.nw)</td>
<td>2</td>
<td>r1.se</td>
<td></td>
</tr>
<tr>
<td>r1.se</td>
<td>(3,r1.sw)</td>
<td>3</td>
<td>r1.sw</td>
<td></td>
</tr>
<tr>
<td>r1.sw</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1.ht</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1.wd</td>
<td>(1,r1.nw), (3,r1.sw)</td>
<td>6</td>
<td>r2.sw</td>
<td></td>
</tr>
<tr>
<td>r2.ht</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.wd</td>
<td></td>
<td>8</td>
<td>r1.wd</td>
<td></td>
</tr>
<tr>
<td>r2.nw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.ne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.se</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ F = [(2,r1.se),(8,r1.wd),(6,r2.sw)] \]

(a)

<table>
<thead>
<tr>
<th>variable</th>
<th>fired structure</th>
<th>eqn_elim</th>
<th>equation</th>
<th>var_elim</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1.nw</td>
<td>1</td>
<td>1</td>
<td>r1.ne</td>
<td></td>
</tr>
<tr>
<td>r1.ne</td>
<td>(1,r1.nw),(2,r1.se)</td>
<td>8</td>
<td>r1.se</td>
<td></td>
</tr>
<tr>
<td>r1.se</td>
<td>(3,r1.sw)</td>
<td>2</td>
<td>r1.sw</td>
<td></td>
</tr>
<tr>
<td>r1.sw</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1.ht</td>
<td>(2,r1.se)</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1.wd</td>
<td>(1,r1.nw), (3,r1.sw)</td>
<td>8</td>
<td>r2.sw</td>
<td></td>
</tr>
<tr>
<td>r2.ht</td>
<td></td>
<td>7</td>
<td>r1.ht</td>
<td></td>
</tr>
<tr>
<td>r2.wd</td>
<td></td>
<td>8</td>
<td>r1.wd</td>
<td></td>
</tr>
<tr>
<td>r2.nw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.ne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.se</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ F = [(7,r1.ht),(8,r1.wd),(6,r2.sw)] \]

(b)

Figure 6: state of the auxiliary data structures and the set F after (a) the sequence [(3,r1.sw), (1,r1.nw)] has been generated and (b) the sequence [(2,r1.se), (3,r1.sw), (1,r1.nw)] has been generated.

```plaintext
delete_equation(e)
    for each v ∈ e do
        if eqn_elim(v) = e then
            let e' represent an equation in v's fired structure
            eqn_elim(v) = e'
            fired(v) = fired(v) - {e'}
        else if e = member(fired(v))
            fired(v) = fired(v) - {e}
```

Figure 7: Algorithm for updating the auxiliary data structures when an equation is deleted from the constraint system.
4.3 Inserting an Equation

If an equation is inserted into the constraint system, the planning algorithm must perform two actions. First, it must remove any reference to the variables in this equation from the auxiliary data structures since they are no longer free variables (the number of equations they belong to is two greater than the number of equations in their fired structures). Second, it must find equations with at least one degree of freedom that can be added to these variable's fired structures so that they once again become free.

The algorithm accomplishes these tasks in the following manner. It examines each of the variables in the equation and processes them differently depending on whether they represent a designated variable or an excess variable. If the variable \( v \) is an excess variable, the equation it eliminates is added to the queue of free equations since 1) the equation retains its degree of freedom (another variable eliminates it); and 2) the equation is not contained in \( v \)'s fired structure so it can be added to \( v \)'s fired structure to help free \( v \).

If the variable \( v \) is a designated variable the planning algorithm checks whether the equation it eliminates belongs to the free equation queue and removes it from the queue if it does. The algorithm takes this action since the equation temporarily loses its degree of freedom. The algorithm then attempts to find an excess variable that eliminates this equation. If it is successful, it makes the excess variable the designated variable for the equation and updates the fired structures of the variables that contain this equation to reflect this fact. It also adds the equation to the queue of free equations since 1) the equation retains its degree of freedom; and 2) the equation is not contained in \( v \)'s fired structure and thus the equation can be added to \( v \)'s fired structure to help free \( v \).

If the algorithm cannot find another variable to eliminate this equation, it removes the equation from all fired structures that contain it and recursively repeats the above procedure for the variables that are associated with these fired structures.

Once the inserted equations have been processed, the planning algorithm hands the free equation queue to the procedure build_sequence which rebuilds the evaluation sequence. Notice that build_sequence does not start from scratch since chunks of the original evaluation sequence have been preserved. The portion of the planning algorithm that handles the insertion of equations is formalized in figure 8.

4.4 Putting It All Together

In many interactive systems it is inconvenient to separate changes into sets of deleted and inserted equations so that they can be processed separately. Instead we must process each change as it is presented and this may involve mixing deletion and insertion operations. In this section we present an algorithm that handles both operations simultaneously.
insert_equation(e)

let F = the set of free equations

for each v ∈ e do
    delete_from_fired_structures(w)
    build_sequence(F)

delete_from_fired_structures(v)

e = eqn_elim(v)
if var_elim(e) = v then
    if e = member(F) then F = F - ((e,v))
    var_elim(e) = find_free_var(e)
    if var_elim(e) != nil then
        F = F ∪ {(e, var_elim(e))}
    for each w ∈ e, w ≠ v do
        if e = member(fired(w)) then
            fired(w) = (fired(w) - ((e,v))) ∪ {(e, var_elim(e))}
        else
            for each w ∈ e, w ≠ v do
                if e = member(fired(w)) then
                    fired(w) = fired(w) - ((e,v))
                    if (eqn_elim(w) != nil) then
                        delete_from_fired_structures(w)
                else
                    F = F ∪ {(e, var_elim(e))}

Figure 8: Algorithm for updating the auxiliary data structures after an equation has been inserted.

One minor change must be made to the deletion algorithm. When processing a variable v in the deleted equation, the algorithm can no longer assume that the equation is either eliminated by the variable or belongs to the variable’s fired structure. The reason this assumption is not valid is that the equation may already have been removed from the auxiliary data structures by the insertion algorithm. If this is the case, the removal of the equation from v’s auxiliary data structures may have caused v to become bound (i.e., not free). Now that the equation is being deleted from the constraint system, v may again become free. Thus the algorithm checks whether the variable’s fired structure is now full (i.e., its size is one less than the number of equations that contain the variable). If it is full, then the variable is free and it can be used to eliminate the equation that does not belong to its fired structure. The algorithm locates this equation and if the equation is not eliminated by any variable, makes this variable the designated elimination variable for the equation, and adds the equation to the list of free equations. The equation is added to the free equation queue since it does not belong to the auxiliary data structures and it needs to
be inserted into these structures. The updated algorithm for a deletion operation is shown in figure 9.

No changes have to be made to the insertion algorithm since each of the operations it performs remain valid. The complete planning algorithm is shown in figure 10. Notice that although the algorithm correctly updates the dependency information, it does not assign order numbers to the equations. It would have been possible to devise a planning algorithm that did this; however, to correctly update these numbers, the algorithm would have to examine more equations than is needed to update the dependency information. To see why, note that the insertion process sometimes makes an excess variable a designated variable. This action may cause the equation that this variable eliminates to change its position in topological order. As a result every equation whose elimination depends directly or indirectly on this equation will also be shifted in topological order. This shift is correctly reflected in the dependency information if the fired structures of the variables that contain this equation are changed to record the equation's new designated variable. However, the order numbers are correctly updated only if the order numbers of all equations whose position in topological order changed are updated. Thus the planning algorithm would have to increase the number of equations it examines if it wanted to assign correct order numbers.

4.5 An Example

Suppose that the equation that equates the heights of rectangles r1 and r2 is removed from the constraint system in figure 1 and that an equation constraining their widths to be equal is inserted instead. That is, remove the equation \( r1.ht = r2.ht \) and insert the equation \( r1.wd = r2.wd \) (labeled equation 9). Suppose the algorithm processes the insertion first and the deletion second. \( r1.wd \) is the designated variable for equation 8 so the planning algorithm searches for an excess variable and finds \( r1.ne \). It updates equation 8's var_elim structure and the fired structures that contain equation 8 to reflect the fact that \( r1.ne \) is the designated variable for equation 8. The algorithm also adds equation 8 to \( F \). The algorithm then turns its attention to \( r2.wd \) and finds that it is an excess variable. Thus it adds the pair \( (6, r2.sw) \) to \( F \) since equation 6 is the equation eliminated by \( r2.wd \) and this equation can be inserted into \( r2.wd \)'s fired structure. The processing of the inserted equation is now completed.

The processing of the deleted equation is equally quick. \( r1.ht \) eliminates the deleted equation so equation 2 is removed from \( r1.ht \)'s fired structure and assigned to its eqn_elim structure. The deleted equation belongs to \( r2.ht \)'s fired structure and it is removed from this structure. Figure 11a shows the values of the auxiliary data structures and the set \( F \) at this point. Notice that the majority of the original information in these structures has been retained. The planning algorithm calls build_sequence to complete the updating of the auxiliary data structures and figure 11b shows their final values.
delete_equation(e)
for each v ∈ e do
  if eqn_elim(v) = e then
    let e' represent an equation in v's fired structure
    eqn_elim(v) = e'
    fired(v) = fired(v) - {e'}
  else if e = member(fired(v))
    fired(v) = fired(v) - {e}
  else if (numeqns(v) - |fired(v)| = 1) then
    e' = find_free_eqn(v)
    eqn_elim(v) = e'
    if var_elim(e') = nil then
      var_elim(e') = v
      F = F ∪ [(e',v)]

Figure 9: Updated algorithm for updating the auxiliary data structures after an equation deletion.

PLAN

let C = the set of changes to the constraint system

F = ∅
for each change ∈ C do
  if change = add_equation then
    insert_equation(e)
  else
    delete_equation(e)
build_sequence(F)

Figure 10: Planning algorithm that incrementally updates the evaluation sequence after the constraint system changes.

4.6 Running Time

The only issue left to consider is the running time of PLAN. PLAN updates the evaluation sequence by changing the designated variables of some of the equations in the constraint network. Let CHANGED denote the set of equations which have different designated variables in the two sequences. Let SUSPECT denote the set of equations that might be suspected of having different designated variables by virtue of having either a direct or indirect dependence on the elimination of one of the variables in the inserted equations. An optimal algorithm will examine O(|CHANGED|) equations since the CHANGE set represents the minimal number of equations that require new designated variables. PLAN examines O(|SUSPECT|) equations since it removes equations from fired structures if their elimination depends either directly or indirectly on the elimination of one of the variables in an inserted equation. In doing so, it invalidates the designated variables for these equations and forces them to be recomputed.
<table>
<thead>
<tr>
<th>variable</th>
<th>fired structure</th>
<th>eqn_elim</th>
<th>equation</th>
<th>var_elim</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1.nw</td>
<td>(1,r1.nw), (2,r1.se)</td>
<td>1</td>
<td>1</td>
<td>r1.nw</td>
</tr>
<tr>
<td>r1.ne</td>
<td>(3,r1.sw)</td>
<td>8</td>
<td>2</td>
<td>r1.se</td>
</tr>
<tr>
<td>r1.se</td>
<td>2</td>
<td>3</td>
<td></td>
<td>r1.sw</td>
</tr>
<tr>
<td>r1.sw</td>
<td>3</td>
<td>4</td>
<td></td>
<td>r2.ne</td>
</tr>
<tr>
<td>*r1.ht</td>
<td>2</td>
<td>5</td>
<td></td>
<td>r2.ht</td>
</tr>
<tr>
<td>*r1.wd</td>
<td>(1,r1.nw), (3,r1.sw)</td>
<td>6</td>
<td></td>
<td>r2.sw</td>
</tr>
<tr>
<td>*r2.ht</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*r2.wd</td>
<td>(4,r2.ne)</td>
<td>8</td>
<td>*8</td>
<td>r1.ne</td>
</tr>
<tr>
<td>r2.nw</td>
<td>(4,r2.ne)</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>r2.ne</td>
<td>(5,r2.ht)</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.se</td>
<td>(5,r2.ht)</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.sw</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ F = ((6,r2.sw),(8,r1.ne)) \]

(a)

<table>
<thead>
<tr>
<th>variable</th>
<th>fired structure</th>
<th>eqn_elim</th>
<th>equation</th>
<th>var_elim</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1.nw</td>
<td>(1,r1.nw), (2,r1.se)</td>
<td>1</td>
<td>1</td>
<td>r1.nw</td>
</tr>
<tr>
<td>r1.ne</td>
<td>(3,r1.sw)</td>
<td>8</td>
<td>2</td>
<td>r1.se</td>
</tr>
<tr>
<td>r1.se</td>
<td>2</td>
<td>3</td>
<td></td>
<td>r1.sw</td>
</tr>
<tr>
<td>r1.sw</td>
<td>3</td>
<td>4</td>
<td></td>
<td>r2.ne</td>
</tr>
<tr>
<td>r1.ht</td>
<td>2</td>
<td>5</td>
<td></td>
<td>r2.ht</td>
</tr>
<tr>
<td>*r1.wd</td>
<td>(9,r2.wd),(1,r1.nw), (3,r1.sw)</td>
<td>8</td>
<td>6</td>
<td>r2.sw</td>
</tr>
<tr>
<td>r2.ht</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*r2.wd</td>
<td>(6,r2.sw),(4,r2.ne)</td>
<td>9</td>
<td>8</td>
<td>r1.ne</td>
</tr>
<tr>
<td>r2.nw</td>
<td>(4,r2.ne)</td>
<td>8</td>
<td>*9</td>
<td>r2.wd</td>
</tr>
<tr>
<td>r2.ne</td>
<td>(5,r2.ht)</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.se</td>
<td>(5,r2.ht)</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2.sw</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b)

Figure 11: (a) Values of the auxiliary data structures and F after insert_equation and delete_equation have processed the equations r1.wd = r2.wd and r1 ht = r2 ht; (b) Values of the auxiliary data structures after build_sequence has been called with the set F as input. The variables whose data structures are affected by these algorithms are marked with asterisks.

It is difficult to further quantify the running time of PLAN since it is difficult to quantify two factors that affect its running time—the number of variables that belong to the set of suspect equations and the distribution of equations eliminated by excess variables (i.e., which equations are eliminated by these variables). If an equation is eliminated by a number of excess variables, it may not have to be removed from any fired structures since
one of these excess variables can be made a designated variable. The operation that makes an excess variable a designated variable requires $O(1|e|)$ work to update the fired structures of the variables associated with the equation where $|e|$ is the number of variables in the equation. If an equation is not eliminated by an excess variable, it must be removed from the fired structures of the variables that it contains. This operation also requires $O(1|e|)$ work. Finally, when REBUILD_SEQ assigns a designated variable to an equation, $O(1|e|)$ work is performed to insert the equation into the fired structures of the variables associated with the equation.

The running time of PLAN is proportional to the number of these operations which are performed. $O(|\text{SUSPECT}|)$ assignments of designated variables are performed and at least one excess variable or fired structure removal operation is performed for each equation in SUSPECT thus leading to another $O(|\text{SUSPECT}|)$ operations. Let $\bar{e}$ denote the average number of variables contained by an equation in the SUSPECT set. Then a lower bound on the running time of PLAN is $O(|\text{SUSPECT}| \bar{e})$.

An upper bound on the running time of PLAN can be obtained by assuming that any variable contained in a suspect equation is an excess variable and that it will eventually become a designated variable. The running time of PLAN is then $O(|\text{SUSPECT}| \bar{e} + \sum_{e \in \text{SUSPECT}} |e|^2)$. This time bound is not realistic however since the number of excess variable operations is likely to be much less than the number of variables in the suspect equations.

In practice, it appears that a constant number of excess variable operations are performed per suspect equation and that an average running time for PLAN is $O(|\text{SUSPECT}| \bar{e})$. In many interactive applications, the number of variables in any equation is bounded by a constant. In this case, the running time of PLAN shortens to $O(|\text{SUSPECT}|)$.

5 Evaluation Strategies

The evaluation sequence established by PLAN defines a partial order on the equations of the constraint network. This partial order can be represented by a directed, acyclic graph. For example, the DAG in figure 12 shows the dependencies induced by the auxiliary data structures in figure 11.

Ideally an evaluator will observe these dependencies as it evaluates equations so that it processes an equation no more than once. That is, if AFFECTED denotes the set of variables that require new values, then the evaluator will ideally reevaluate only $O(|\text{AFFECTED}|)$ equations. Unfortunately, this goal is easier stated than achieved. Since PLAN does not associate order numbers with the equations, it is difficult to determine a priori whether one equation precedes another.
In this section we discuss several approaches to the problem, including naive propagation, nullification/reevaluation, and approximate topological ordering. None of these approaches achieve the optimal time bound, although in a separate paper, we present an evaluator that achieves this time bound as a side effect of trying to minimize the number of reevaluated equations.

5.1 Naive Propagation

Naive propagation ignores the problem of reevaluating equations more than once. It keeps a work set of potentially unsatisfied constraints and at each step, it arbitrarily removes a constraint from this set, solves it for the designated variable, and if the variable's value changes, adds the equations that the variable belongs to to the work set. The advantage of this approach is that propagation is dependent on the values of the variables and thus it quiesces whenever a variable's value is unchanged. The disadvantage of this approach is that the algorithm's running time can be nonlinear in the size of AFFECTED. Indeed, its worst case behavior is exponential. An example of a problem on which naive propagation requires exponential time is presented in [Reps 83].

5.2 Nullification/Reevaluation

Nullification/reevaluation works by reevaluating any variable whose value depends directly or indirectly on one of the designated variables in an initially unsatisfied equation. As suggested by the name, this strategy has two phases. The first phase nullifies the values of the designated variables in the initial set of unsatisfied equations and then recursively traverses the dependency graph, nullifying the values of all variables that depend on these nullified variables. The second phase then propagates new values throughout the dependency graph [Reps 83].

Both of these phases traverse the portion of the dependency graph reachable from the initial set of inconsistent variables. Let INFLUENCED denote the set of variables that belong to this portion of the DAG. Then the total amount of work done by this algorithm is
O(|INFLUENCED|).

The advantage of this algorithm is that it is linear in the number of equations reevaluated. The disadvantage of this algorithm is that propagation is dependent on the structure of the dependency graph and not the values of the variables. Thus it will not quiesce if the value of some variable is unchanged. As a consequence, the size of |INFLUENCED| could be much greater than the size of |AFFECTED|, leading to the needless evaluation of a great many equations.

5.3 Approximate Topological Ordering

The method of reevaluating constraints using approximate topological ordering was pioneered by Roger Hoover [Hoover 86, Hoover 87]. This approach assigns order numbers to equations based on their estimated position in topological order. These order numbers are continuously updated during the planning and evaluation phases as information is gathered that allows their positions to be identified more precisely. Since the order numbers are only best guesses as to where an equation is located in topological order, an evaluator may satisfy an equation more than once. However, since the order numbers are continuously updated, it is expected that these guesses will be fairly accurate and that the number of multiply evaluated equations will be minimal. Indeed, Hoover reports that in practice, the number of reevaluated equations is only a few percent greater than the size of AFFECTED [Hoover 86].

This approach can be adapted to the problem at hand in the following manner. Assign order numbers to the variables so that variables that appear earlier in the topological order have higher values and variables that appear later in the order have smaller values. Intuitively, these values can be thought of as distances from the sink nodes. Since variables that appear earlier in topological order tend to be farther from the sink nodes, they deserve higher values.

During the evaluation phase, the evaluator maintains a work set of potentially unsatisfied variables. At each step it removes the highest numbered variable from the work set, evaluates it, and if its value is changed, adds the equations that contain this variable to the work set. The evaluator also ensures that the order number of the variable is greater than the order numbers of any of its successors by assigning to its order number the sum of the order numbers of its successors plus the number of its successors (order_number(v) = \sum_{w \in succ(v)} order(w) + |succ(v)| where succ(v) denotes the successors of v in topological order).

The planning algorithm also updates the order numbers by computing the order number of a variable whenever it is freed and made to eliminate an equation. Figures 13 and 14 show the new versions of PLAN and EVALUATE.

The advantage of this algorithm is that it performs propagation based on both the values of the variables and an educated guess as to which variables to evaluate next. Thus it tends to evaluated O(|AFFECTED|) equations.
insert_equation(e)
   for each v \in e do
      e' = eqn_elim(v)
      if var_elim(e') = v then
         var_elim(e') = find_free_var(e')
      if var_elim(e') \neq nil then
         F = F \cup ((e', var_elim(e'))
         for each w \in e', w \neq v do
            if e' = member(fired(w)) then
               fired(w) = (fired(w) - [(e', v)]) \cup [(e', var_elim(e'))]
            else
               for each w \in e', w \neq v do
                  if e' = member(fired(w)) then
                     fired(w) = fired(w) - [(e', v)]
                  if (eqn_elim(w) \neq nil) then
                     delete_from_fired_structures(w)
            else
               F = F \cup [(e', v)]

delete_equation(e)
   for each v \in e do
      if eqn_elim(v) = e then
         let e' represent an equation in v's fired structure
         eqn_elim(v) = e'
         fired(v) = fired(v) - [e']
      else if e = member(fired(v))
         fired(v) = fired(v) - [e]
      else if (numeqns(v) - |fired(v)| = 1) then
         fix_order_numbers(v)
         e' = find_free_eqn(v)
         eqn_elim(v) = e'
         if var_elim(e') = nil then
            var_elim(e') = v
            F = F \cup [(e', v)]

build_sequence(F : set)

   while F \neq \emptyset do
      (e, v) = delete_min_element(F)
      for each w \in e, v \neq w do
         if (numeqn(w) - |fired(w)|) > 1 and e \neq member(fired(w)) then
            fired(w) = fired(w) \cup [(e, v)]
         if (numeqn(w) - |fired(w)|) = 1 then
            fix_order_numbers(w)
            e' = find_free_eqn(w)
            eqn_elim(w) = e'
            if solved(e') = false then
               F = F \cup [(e', w)]
               solved(e') = true
            var_elim(e') = w

Figure 13: Implementation of approximate topological ordering in the planning algorithms.
\textbf{EVALUATE}(E : priority_queue)

\[ E = \text{set of initially inconsistent equations} \]

\textbf{while} \( E \neq \emptyset \) \textbf{do}
\[ (e,v) = \text{delete_max_element}(E) \]
\[ \text{fix_order_numbers}(v) \]
\[ \text{value}(v) = \text{solve_eqn}(e,v) \]
\[ \text{if} \ \text{value}(v) \neq \text{old_value}(v) \ \text{then} \]
\[ \text{for each} \ e' \in \text{eqn}(v), \ e' \neq e \ \text{do} \]
\[ E = E \cup \{(e', \text{var_elim}(e'))\} \]

\textbf{fix_order_numbers}(v)

\[ \text{order}(v) = |\text{fired}(v)| \]
\[ \text{for each} \ w \in \text{fired}(v) \ \text{do} \]
\[ \text{order}(v) = \text{order}(v) + \text{order}(w) \]

Figure 14: Implementation of approximate topological order in the evaluation algorithms.

The drawback of this approach is that it is still naive propagation augmented with an educated guess mechanism and thus in the worst case it can reevaluate an exponential number of equations.

\textbf{6 Experimental Results}

The incremental planning algorithm and the evaluation algorithms based on naive propagation and approximate topological ordering have been implemented in a prototype system called CONSTRAINT [Vander Zanden 88b, 88c]. For comparison purposes we have also implemented a planning algorithm that nonincrementally constructs an evaluation sequence based on the techniques used in ThingLab [Bornig 81,86].

CONSTRAINT is a generator of graphical applications that takes a specification of an application as input and implements it as a mouse and menu-based system. The specification uses multilinear constraints to describe the graphical layout of objects and the system uses a noncircular, multilinear constraint solver to satisfy these constraints as the user performs editing operations. Figures 15 and 16 show how the constraint solver performed in an application that manipulated binary trees. The figures show the number of equations examined and evaluated by the constraint solver in a session in which a binary tree was constructed and then manipulated by randomly swapping and deleting children.

Figure 15 shows the number of equations that the incremental and nonincremental planning algorithm examined during each operation. For comparison purposes, the average number of equations present in the constraint network during each of these operations is also shown. Figure 16 compares the number of equations solved by an evaluator using naive propagation and approximate topological ordering. In these figures, operations 1-13 constructed the tree by creating and adding large subtrees, operations 14-18 swapped children, and operations 19-21 deleted children.
Figure 15: Number of equations examined by the incremental and nonincremental planning algorithms for each operation of a sample editing session on binary trees.

In this editing session, the number of equations examined by incremental planning was independent of the number of equations in the constraint system whereas the number of equations examined by the nonincremental planner was proportional to the number of equations in the system. As a consequence, once a steady state had been achieved, incremental planning examined 5-7 times fewer equations. This difference became quite noticeable during the swap children operations when the nonincremental planner caused an irritating delay in the updating of the display. In contrast, the display was immediately updated when the incremental planning algorithm was employed.

Figure 16 shows how approximate topological ordering dominated naive propagation during this editing session. Approximate topological ordering evaluated fewer equations in each operation, sometimes evaluating only half as many equations as naive propagation. It also gave a much more uniform performance. While the number of equations evaluated by approximate topological ordering increased uniformly during the tree creation process, the number of equations evaluated by naive propagation varied erratically and unpredictably.
Figure 16: Number of equations that were solved during each operation of a sample editing session on binary trees by an evaluator using naive propagation and approximate topological ordering.

These results are typical of the performance of the planning and evaluation algorithms. In general, the nonincremental planning algorithm examines every equation that is reachable from the modified set of equations while the number of equations examined by the incremental planning scheme is independent of this region's size. Of course, since the running time of the incremental planning scheme is $O(|\text{SUSPECT}| \cdot |\ell|)$, it may process each equation $|\ell|$ times. Thus whether it outperforms nonincremental planning in practice depends on the size of $|\ell|$, the size of SUSPECT, and the number of equations in the connected region. In the CONSTRAINT system, the average number of variables per constraint is roughly 3 and thus incremental planning is preferable, except during the initial start up period when the number of equations in the system is fairly small.

The approximate topological ordering scheme has worked out extremely well in practice. Although we have not been able to compare the number of equations it evaluates with the size of AFFECTED (measuring the size of AFFECTED is very difficult since the very act of assigning exact order numbers tends to change the dependency information),
the number of equations it evaluates is proportional to the size of the change introduced by
the user and thus its performance is quite satisfactory.

7 Conclusions

In this paper we have presented a planning algorithm that incrementally orders the
equations in a noncircular, multilinear constraint system and an evaluation algorithm
that solves equations based on the partial order defined by this planning algorithm. As
shown in section 6, the planning algorithm can achieve sizeable reductions in the number
of equations examined during the planning phase of constraint satisfaction. The results
in this section are typical of the performance of the planning and evaluation algorithms in
practice.

These results are quite promising and suggest that an incremental approach to
planning should be pursued in noncircular, multilinear constraint systems. In a
companion paper, we present a heuristic that tries to guide this planning process so that a
minimal number of equations are solved during the evaluation phase [Vander Zanden
88a]. This heuristic has the beneficial side-effect of assuring that the evaluator always
solves a polynomial number of equations.

Acknowledgements

Dexter Kozen, Tim Teitelbaum, Bill Pugh, Charles Elkan, and Roger Hoover all
provided helpful comments and advice that assisted the development of the ideas in this
paper.

References

[ Borning 81] Borning, A. 1981. The programming language aspects of ThingLab, a
constraint-oriented simulation laboratory. ACM Transactions on Programming
Languages and Systems, 3:357-387.


[ Dennis 83] Dennis, J.E., Jr. and Schnabel, R.B. Numerical Methods for Unconstrained

application to financial modelling. Software—Practice and Experience, 14(12):1169-
1181.

in the Animus System. PhD Dissertation, University of Washington. Also available
as technical report 86-09-01.


Carnegie-Mellon University, Pittsburg, PA.


