Solving Difference Constraints Incrementally

G. Ramalingam, J. Song[*] and L. Joscovicz
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598
R. E. Miller
Dept of Computer Science, University of Maryland, College Park, MD 20742

October 27, 1995

Abstract

A system of difference constraints consists of a set of inequalities of the form $x_i - x_j \leq b_{i,j}$. Such systems occur in many applications, *e.g.*, temporal reasoning. The best known algorithm to determine if a system of difference constraints is feasible (*i.e.*, if it has a solution) and to compute a solution if the system is feasible runs in $\Theta(mn)$ time, where n is the number of variables and m is the number of constraints.

In this paper, we explore the problem of maintaining a solution to a system of difference constraints as the system undergoes changes such as the addition or removal of constraints. The problem arises in the context of an interactive system that allows users to model the temporal behavior of a multimedia application as a system of difference constraints. We present an incremental algorithm for the problem (which enables immediate feedback to the user as and when a constraint added or modified by the user creates an infeasible system.)

Our algorithm processes the addition of a constraint in time $O(m + n \log n)$ and the removal of a constraint in constant time, as long as the original system is feasible. (The time taken to process the addition of a constraint can be more precisely described as $O(\|\Delta\| + |\Delta| \log |\Delta|)$, where $|\Delta|$ denotes the number of variables whose values are changed to compute the new feasible solution, and $\|\Delta\|$ denotes the number of constraints involving the variables whose values are changed.) If the original system is infeasible, the algorithm processes any change in $O(m + n \log n)$ *amortized* time.

The algorithm we present can also be used to check for the existence of negative cycles in a dynamic (*i.e.*, changing) graph.

[*]Current address: Dept of Computer Science, University of Maryland, College Park, MD 20742

1 Introduction

A system of difference constraints consists of a set of inequalities of the form $x_i - x_j \leq b_{i,j}$. Such a system is said to be feasible if there exists a solution to the system of inequalities. Systems of difference constraints have been used in various application areas, most notably for temporal reasoning in both AI and real-time systems area. Dechter et al. [5, 14] have formulated a unifying temporal reasoning framework, called temporal constraint network, using this type of constraints. Many real-time programming languages[10, 15, 13, 24] provide language constructs that allow the programmer to express difference constraints. Recently, difference constraints have also been proposed as a way of specifying the temporal behavior of multimedia applications[12, 3]. In multimedia applications, for example, the play duration of a multimedia "object" (an audio or video segment, for example) has minimum and maximum bounds. There may also be constraints on when different "object"s are to be played relative to each other. All of these constraints can be expressed using a system of difference constraints.

In interactive systems that let users create such systems it is useful to check for the feasibility of the system incrementally, and provide feedback to the user when a constraint is added or modified by the user that creates an infeasible system.

A system of difference constraints can be represented by a weighted directed graph, called a constraint graph, such that the system is feasible iff there exists no negative weight cycle in the constraint graph. Further, the solution to the single-source shortest-paths problem on the constraint graph, with respect to a distinguished *source* vertex yields a feasible solution to the system of constraints. Since the constraint graph will contain negative length edges in general (otherwise, the graph cannot have a negative length cycle, and the system is trivially feasible!), the standard algorithms for testing a graph for negative cycles (and for computing the lengths of the shortest paths) take $O(mn)$ time, where m is the number of edges (constraints) in the graph, and n is the number of vertices (variables) in the graph.

In this paper, we present an incremental algorithm for testing feasibility of a system of difference constraints that processes the insertion of a constraint in time $O(m + n \log n)$ and the deletion of a constraint in $O(1)$ time. The algorithm also maintains a feasible solution to the system of constraints. The algorithm uses an adaptation of an incremental algorithm for the shortest-paths problem, and is expected to take less time than the worst-case complexity of $O(m + n \log n)$. A more precise characterization of the running-time of the algorithm is $O(\|\Delta\| + |\Delta| \log |\Delta|)$, where $|\Delta|$ denotes the number of variables whose values are changed to compute to the new feasible solution, and $\|\Delta\|$ denotes the number of constraints involving these variables (whose values are changed). Though our algorithm is an adaptation of an incremental algorithm for the shortest-paths problem, it does *not* maintain the shortest-paths solution — we will see that though it would be correct to do so, that turns out to be unnecessary.

The rest of the paper is organized as follows. In Section 2 we present some preliminary material concerning systems of difference constraints. In Section 3 we outline the problem addressed in this paper. In Section 4 we present a simple algorithm for the problem, which will help motivate an improved algorithm presented in Section 5. We assume in Sections 4 and 5 that the original system is feasible. In Section 6 we show how to handle infeasible systems. In Section 7 we discuss some previous related work. In Section 8 we discuss possible future work.

2 Preliminaries

A system of difference constraints $< V,\ C >$ consists of a set V of variables and a set C of linear inequalities of the form $x_i - x_j \leq b_{i,j}$, where $x_i, x_j \in V$. A feasible solution for a system of difference constraints is an assignment of real values to the variables (a function from the variables to reals) that satisfies all the given constraints. A system is said to be feasible iff it has a feasible solution.

A directed, weighted graph G = <V, E, *length* > consists of a set of vertices V, a set of edges E, and a function *length* from E to reals. We will denote an edge from vertex u to vertex v by $u \rightarrow v$. We will denote the length of a shortest path from u to v in a weighted graph G by $dist_G(u, v)$.

Pratt[17] has shown how a system of difference constraints $< V, C >$ can be represented by a directed, weighted graph G = <V, E, *length* > whose vertices correspond to variables, and whose edges correspond to constraints.

Definition 1 (Constraint Graph) *The constraint graph of a system of difference constraints* $< V, C >$ *is a directed, weighted graph* $G = < V,\ E,\ length >$ *where*

$$\begin{array}{l} E = \{\ x_j \rightarrow x_i \mid x_i - x_j \leq a_{i,j} \in C\ \} \\ length(x_j \rightarrow x_i) = a_{i,j} \qquad \text{iff } x_i - x_j \leq a_{i,j} \in C \end{array}$$

Thus, given a constraint graph, $< V, E, length >$, the corresponding constraint system is $< V, \{\ v - u \leq length(u \rightarrow v) \mid u \rightarrow v \in E\} >$.

For convenience, we will assume that a system of constraints contains at most one inequality per ordered pair of variables: that is, the system cannot include two constraints $x - y \leq a$ and $x - y \leq b$, where $a \neq b$. If there are multiple constraints on the same ordered pair of variables, then the constraint graph is a multigraph: that is, multiple edges may exist between the same pair of vertices. Except for the fact that an edge is no longer determined by its endpoints, everything discussed in this paper applies equally well to such multigraphs. (Alternatively, one may define $length(x_j \rightarrow x_i)$ to be min $\{c \mid x_i - x_j \leq c \in C\}$. In the context of our incremental algorithm, the length of an edge can be maintained by storing all the corresponding constraints as a heap (priority queue) associated with the edge.)

While the constraint graph defined above adequately describes the system of constraints, it is useful to augment this graph with an extra source vertex for purposes of computing a feasible solution.

Definition 2 (Augmented Constraint Graph) *The augmented constraint graph of a system of difference constraints $< V, C >$ is a directed, weighted graph $G' = < V', E', length' >$ where*

$$\begin{array}{ll} V' = V \cup \{src\} & \text{where } src \notin V \\ E' = \{\ x_j \rightarrow x_i \mid x_i - x_j \leq a_{i,j} \in C\ \} \cup \{src \rightarrow x_i | x_i \in V\} & \\ length'(x_j \rightarrow x_i) = a_{i,j} & \text{if } x_i - x_j \leq a_{i,j} \in C \\ length'(src \rightarrow v_i) = 0 & \text{for } v_i \in V \end{array}$$

Theorem 1 ([17]. See also [4]) *A system of difference constraints is consistent if and only if its augmented constraint graph has no negative cycles if and only if its constraint graph has no negative cycles.*

Theorem 2 ([17]) *Let G be the augmented constraint graph of a consistent system of constraints $< V, C >$. Then D is a feasible solution for $< V, C >$, where*

$$D(u) = dist_G(src, u)$$

3 The Problem

We are interested in maintaining a feasible solution to a system of constraints as it undergoes changes. In the first part of the paper we will assume that the original system of constraints is itself feasible. The possible changes to the system are: the deletion or addition of a new constraint, the modification of an existing constraint, or the addition or deletion of a variable.

The addition or deletion of a variable is handled easily. We update the constraint graph, and, in the case of a new (unconstrained) variable, we initialize its value to be zero. The system continues to be feasible.

The deletion of a constraint can not introduce infeasibility, since the system becomes less constrained. The original solution continues to be a feasible solution of the new system of constraints as well. Similarly, the relaxation of a constraint, which corresponds to the increase in the length of the corresponding edge in the constraint graph, does not affect the solution. Hence, such changes can be processed in constant time: just update the constraint graph, by removing or changing the length of the appropriate edge, and leave the values of variables alone.

The change that can affect the feasibility of the system is the addition of a new constraint $x_i - x_j \leq b$, which corresponds to the insertion of an edge from x_j to x_i whose length, denoted by $length(x_j \rightarrow x_i)$, is b. This is the problem that will be considered in the remaining parts of the paper. (Tightening an existing constraint, which corresponds to a decrease in the length of an existing edge in the constraint graph, can be similarly handled.) Theorem 2 suggests that we recompute shortest-path distances from *src* in the modified constraint graph and use that as the new feasible solution. The algorithm presented in Section 4 does precisely that. However, as we will see in Section 5, this turns out to be more work than is necessary — we will see how it may often be easier to compute a feasible solution that is *not* the shortest-paths solution.

4 A Simple $O(m + n \log n)$ Algorithm

We first present a simple algorithm that processes the addition of a new constraint in $O(m + n \log n)$ time. While this algorithm is already better than the batch algorithm for computing a feasible solution, it is presented primarily to motivate an improved algorithm we present for the problem in the next section. Hence, we will just present the algorithm informally.

The algorithm has to perform two tasks: it has to determine if the addition of the new constraint makes the system infeasible; if not, it also has to compute a feasible solution to the new set of constraints.

Checking For Feasibility

Adding a new constraint $v - u \leq c$ amounts to inserting a new edge $u \rightarrow v$ of length c to the constraint graph. Let us denote the original constraint graph by G and the new constraint graph by G'. From Theorem 1 we know that adding the new constraint will introduce infeasibility iff there is a negative cycle in the new constraint graph G'. We know that G does not have any negative cycle (from the feasibility of the original constraint system). Hence, if G' has any negative cycle, the cycle must involve the new edge $u \rightarrow v$. Obviously, there exists a negative cycle in the graph involving edge $u \rightarrow v$ iff $dist_G(v, u) + c < 0$. (Recall that $dist_G(v, u)$ denotes the length of the shortest path from v to u in the original graph G.) Hence, the problem reduces to that of computing $dist_G(v, u)$.

Now, in general, the graph G will contain edges of negative length. Consequently, computing $dist_G(v, u)$ using the standard shortest path algorithms can take $O(mn)$ time. It turns out that we can do better by using the feasible solution for the original set of constraints. The idea is to adapt a technique due to Edmonds and Karp for transforming the length of every edge to a non-negative real without changing the graph's shortest paths [7, 22]

Theorem 3 (Edmonds and Karp[7]) *Let $G = < V, E, length >$ be a directed, weighted graph. Let f be a real valued function on V, the set of vertices. Define a new weighted graph G_f, the graph G scaled by f, as follows: $G_f =< V, E, length_f >$, where $length_f$ is defined by*

$$length_f(x, y) = f(x) + length(x \rightarrow y) - f(y)$$

A path P from x to y is a shortest path in G if and only if it is a shortest path in G_f. Further, the lengths of the shortest paths under the two weight functions are related by:

$$dist_{G_f}(x, y) = f(x) + dist_G(x, y) - f(y)$$

Now, consider what happens if we scale the original constraint graph G by any feasible solution D. The new length of an edge $x \rightarrow y$ will be $D(x)+length(x \rightarrow y)-D(y)$, which is non-negative since D is a feasible solution for C. (Recall that the edge $x \rightarrow y$ corresponds to the constraint $y - x \leq length(x \rightarrow y)$.) This implies that we can compute the lengths of shortest paths in graph G_D using Dijkstra's algorithm in $O(m + n \log n)$ time, from which we can directly compute the lengths of shortest paths in G using the above theorem. In particular, we can compute $dist_G(v, u)$ in $O(m + n \log n)$ time and determine if the new system is feasible.

Computing a Feasible Solution

If we determine that adding the new constraint does not make the system infeasible, we then need to find a feasible solution to the new set of constraints. From Theorem 2, we know that the lengths of shortest paths from the source vertex in the new augmented constraint graph can be used as a feasible solution. We can also use the technique outlined above to scale edge lengths so that they are non-negative. This lets us use Dijkstra's algorithm to compute the shortest paths, and hence, a feasible solution in $O(m + n \log n)$ time.

There are, however, a couple of issues. First, note that we now are computing shortest paths in the augmented constraint graph. For the transformed edge lengths to be non-negative for the extra edges from the source vertex, the initial feasible solution must also satisfy the additional "constraints" implied by these extra edges. As long as the feasible solution was originally computed using Theorem 2 and subsequentially updated using the algorithm in this section, these additional constraints will be satisfied. Second, note that while the scaled length is guaranteed to be non-negative in G_D for every edge that was in the original graph, this need not hold for the newly inserted edge $u \rightarrow v$. However, this is not a problem. Shortest paths lengths in G' can still be computed in $O(m + n \log n)$ time by doing two shortest-paths calculations in G (each of which, in turn, is performed using a shortest-path calculation on the scaled graph G_D and "unscaling" using Theorem 3), as follows.

$$dist_{G'}(src, x) = min(dist_G(src, x), dist_G(src, u) + length(u \rightarrow v) + dist_G(v, x)) \qquad (1)$$

The above equation follows by considering both the shortest path from src to x that does not contain edge $u \rightarrow v$ and the shortest path from src to x that contains edge $u \rightarrow v$.

5 An Improved Algorithm

A Different Feasible Solution

The algorithm outlined in the previous section does not fully utilize the original feasible solution in computing the new feasible solution. It uses the original feasible solution only to transform edge-lengths to be non-negative, and after that computes shortest-path lengths from scratch. We now show that changing the right-hand side of equation 1 lets us use the previous feasible solution more effectively in computing a new feasible solution.

Let $< V, C >$ denote the original system of constraints, and let D be a feasible solution for $< V, C >$. Let C' denote the new set of constraints $C \cup \{v - u \leq c\}$. Define D' as below[1]

$$D'(x) = min(D(x), D(u) + length(u \rightarrow v) + dist_G(v, x)) \qquad (2)$$

Let us say a vertex x is "affected" if $D'(x) \neq D(x)$.

Theorem 4 *$< V, C' >$ is feasible iff u is not affected.*

Proof
We have $D'(u) = min(D(u), D(u) + length(u \rightarrow v) + dist_G(v, u))$. Thus u is affected iff $D'(u) \neq D(u)$ iff $D(u) > D(u) + length(u \rightarrow v) + dist_G(v, u)$ iff $length(u \rightarrow v) + dist_G(v, u) < 0$ iff the new constraint graph has a negative cycle iff $< V, C' >$ is infeasible.
$\square$

Theorem 5 *If $< V, C' >$ is feasible then D' is feasible solution for $< V, C' >$.*

Proof
(a) We first show that D' satisfies every constraint in C. Assume that the constraint is represented by the edge $x \rightarrow y$. We consider two cases.

Case 1: $D'(x) = D(x)$. Then, we have

$$\begin{array}{lll} D'(y) & \leq D(y) & \text{(from the definition of } D'\text{)} \\ & \leq D(x) + length(x \rightarrow y) & \text{(since D is feasible)} \\ & \leq D'(x) + length(x \rightarrow y) & \text{(by assumption)} \end{array}$$

Case 2: $D'(x) = D(u) + length(u \rightarrow v) + dist_G(v, x)$.

$$\begin{array}{lll} D'(y) & \leq D(u) + length(u \rightarrow v) + dist_G(v, y) & \text{(from definition of } D'\text{)} \\ & \leq D(u) + length(u \rightarrow v) + dist_G(v, x) + length(x \rightarrow y) & (dist \text{ property}) \\ & \leq D'(x) + length(x \rightarrow y) & \end{array}$$

[1]The motivation to consider this comes from the desire to use an incremental algorithm to update the feasible solution. Assume, for the moment, that the previous feasible solution is exactly the shortest-path distance from *src*; that is, $D(x) = dist_G(src, x)$. In this case, the right-hand side of equation 1 reduces to the right-hand side of equation 2, which is easier to compute since it requires the computation of only $dist_G(v, x)$. However, given our strategy for handling constraint deletions — namely, delete the appropriate edges of the constraint graph but do not change the feasible solution — the current feasible solution is not, in general, the shortest-path distance from *src*. However, it turns out that the right-hand side of equation 2 can still be used as a feasible solution for the new system, even though it may not be the shortest path distance from *src*.

(b) We now show that D' satisfies the new constraint $u \rightarrow v$. Since $< V, C' >$ is feasible, we know that u is unaffected (from Theorem 4). That is, $D(u) = D'(u)$. Now,

$$\begin{aligned}
D'(v) &= min(D(v), D(u) + length(u \rightarrow v) + dist_G(v,v)) \\
&= min(D(v), D(u) + length(u \rightarrow v)) \\
&\leq D(u) + length(u \rightarrow v) \\
&\leq D'(u) + length(u \rightarrow v)
\end{aligned}$$

Hence, the new constraint $v \leq u + length(u \rightarrow v)$ is satisfied. □

Theorems 4 and 5 show how we can process the addition of the new constraint. We can compute D' and check if u is affected. If u is affected, the new system is infeasible. If u is not affected, we know that the new system is feasible, and that D' is a feasible solution to the new system. Note that the new feasible solution D' may not be the same feasible solution ($dist_{G'}(src, x)$) computed by our original algorithm. We now show how D' can be computed efficiently.

Computing D'

Theorem 6 *Let x be the parent of y in some shortest path tree for v in the original graph. If x is unaffected, then y is also unaffected.*

Proof

$$\begin{aligned}
D(y) &\leq D(x) + length(x \rightarrow y) \\
&\quad \text{(since D is feasible)} \\
&\leq D(u) + length(u \rightarrow v) + dist_G(v,x) + length(x \rightarrow y) \\
&\quad \text{(since } x \text{ is unaffected, and hence} \\
&\quad D(x) \leq D(u) + length(u \rightarrow v) + dist_G(v,x)) \\
&\leq D(u) + length(u \rightarrow v) + dist_G(v,y) \\
&\quad \text{(since } x \text{ is } y\text{'s parent in the shortest-path tree, and hence} \\
&\quad dist_G(v,y) = dist_G(v,x) + length(x \rightarrow y))
\end{aligned}$$

Hence y is also unaffected. □

Let us say that an edge $x \rightarrow y$ is affected iff vertex x is affected. Let H denote the subgraph of G consisting only of affected edges. Thus, $H = < V, E_a, length|E_a >$, where E_a is the set of affected edges and $length|E_a$ denotes the restriction of $length$ to E_a.

Theorem 7

$$D'(x) = min(D(x), D(u) + length(u \rightarrow v) + dist_H(v,x))$$

Proof
Obviously, $dist_H(v,x) \geq dist_G(v,x)$. Theorem 6 implies that $dist_H(v,x) = dist_G(v,x)$ for all affected vertices x. The result follows. $\Box$

The algorithm in Figure 5 shows how we can utilize the results presented so far to efficiently to identify if the new system is feasible and, if it is, to compute a new feasible solution, namely D'. The algorithm is essentially the application of Dijsktra's algorithm (see [4], for example) to the graph H consisting only of affected edges. Our algorithm makes use of the following heap operations. The operation $InsertIntoHeap(H, i, k)$ inserts an item i into heap H with a key k. The operation $FindAndDeleteMin(H)$ returns the item in heap H that has the minimum key as well as its key and deletes the item from the heap. The operation $AdjustHeap(H, i, k)$ inserts an item i into $Heap$ with key k if i is not in $Heap$, and decreases the key of item i in $Heap$ to k if i is in $Heap$. The function $KeyOf(H, i)$ returns the key of item i in heap H if i is in H and ∞ otherwise.

Lines [4-7] and [15-20], by themselves, would apply Dijkstra's algorithm to the graph G, with edge lengths scaled by D. Line [8] excludes edges that are not affected from consideration, effectively resulting in the algorithm being applied to graph H, with edge lengths scaled by D. Line [9] checks for feasibility of the new system (using Theorem 4).

Let us now analyze the complexity of the algorithm in Figure 1. Observe that the `while` loop in the algorithm iterates once for every affected variable, that is, once for every variable whose value is changed in order to compute the new feasible solution. The inner `for` loop processes the constraints involving the affected variable x. Consequently, the number of `FindAndDeleteMin` operations performed is $|\Delta|$ and the number of `AdjustHeap` operations performed is $\|\Delta\|$. The complexity of the algorithm depends on the type of heap we use. We assume that `PriorityQueue` is implemented as a relaxed heap [6]. Both insertion of an item into a relaxed heap and decreasing the key of an item in a relaxed heap cost constant time, while finding and deleting the item with the minimum key costs $O(\log p)$ time, where p is the number of items in the heap. Thus, each `FindAndDeleteMin` operation runs in time $O(\log |\Delta|)$ time and each `AdjustHeap` takes constant time. Hence, the algorithm will run in $O(\|\Delta\| + |\Delta| \log |\Delta|)$ time.

Implementing the priority queue as an AF heap[8] will improve the complexity of the algorithm to $O(\|\Delta\| + |\Delta| \log |\Delta| / \log\log |\Delta|)$.

6 Handling Infeasible Systems

We have so far assumed that the constraint system being modified is a feasible one. If the constraint being added would result in infeasibility, the new constraint is rejected. However, it may be useful, in some situations, to allow the addition of constraints that cause the system to become infeasible. We now discuss how our algorithm can be adapted to do this. (See the algorithms in Figures 2 and 3.)

```
function AddToFeasible (<V,E, length>, $v - u \leq c$, D)
parameters
      <V,E, length>:  a feasible constraint graph
      D : a feasible solution for <V,E, length>
      $v - u \leq c$:  a new constraint to be added
local variables
      PriorityQueue:  a heap of vertices
      D':  a map from vertices to reals
begin
[1]   Add edge $u \rightarrow v$ to E
[2]   length($u \rightarrow v$) := c
[3]   D' := D
[4]   PriorityQueue := $\phi$
[5]   InsertHeap(PriorityQueue, v, 0)
[6]   while PriorityQueue != $\phi$ do
[7]      (x, dist_x) := FindAndDeleteMin(PriorityQueue)
[8]      if $D(u) + length(u \rightarrow v) + dist\_x < D(x)$ then
[9]         if $(y = u)$ then
[10]           /* Infeasible system:  reject new constraint */
[11]           remove edge $u \rightarrow v$ from E
[12]           return false
[13]        else
[14]           $D'(x)$ := $D(u) + length(u \rightarrow v) + dist\_x$
[15]           for every vertex y in Succ(x) do
[16]              scaledPathLength := $dist\_x + (D(x) + length(x \rightarrow y) - D(y))$
[17]              if $(scaledPathLength < KeyOf(PriorityQueue, y))$
[18]                 AdjustHeap(PriorityQueue, y, scaledPathLength)
[19]              fi
[20]           od
[21]        fi
[22]     fi
[23]  od
[24]  /* Feasible system:  Update solution */
[25]  D := D'
[26]  return true
end
```

Figure 1: Algorithm to update a feasible constraint system after the addition of a constraint. Function `AddToFeasible` returns `false` if adding constraint $v - u \leq c$ to a feasible system $< V, E, length >$ would introduce infeasibility. Otherwise, it adds the new constraint to the system, updates the solution D to the system of constraints, and returns `true`.

```
function AddConstraint (<V,E, $length$>, UnProcessed, $v-u \leq c$, $D$)
declare
   <V,E, $length$>:  a feasible constraint graph
   $D$ :  a feasible solution for <V,E, $length$>
   UnProcessed:  a set of constraints
   $v-u \leq c$:  a new constraint to be added
begin
   if (UnProcessed is empty) then
      /* System is feasible before addition of new constraint */
      if ( not AddToFeasible (<V,E, $length$>, $v-u \leq c$, $D$) then
         /* Adding new constraint will cause infeasibility */
         UnProcessed := { $v-u \leq c$ }
      fi
   else
      /* System is infeasible before addition of new constraint */
      Add $v-u \leq c$ to UnProcessed
   fi
end
```

Figure 2: Algorithm to update the constraint system after the addition of a constraint.

We modify the algorithm so that it maintains a partition of the set of constraints into two sets: a feasible set of constraints, which is represented by a constraint graph and for which a feasible solution is maintained, and a set, which we call *UnProcessed*, of the remaining constraints. If the whole system is feasible, then *UnProcessed* is guaranteed to be empty. When a new constraint is added to a feasible system, we determine if the addition of a new constraint introduces infeasibility. If not, we process it as before by adding it to the system and updating the feasible solution as necessary. If the new constraint causes infeasibility, we leave the constraint graph and the feasible solution unmodified, and initialize the set *UnProcessed* to consist of the new constraint.

Once the systems becomes infeasible, the addition of a new constraint cannot make the system feasible. Hence, the addition of a new constraint is processed trivially, by just adding the new constraint to the set *UnProcessed*.

When a constraint is deleted, we check if it is in the constraint graph or the set *UnProcessed*. If it is in the set *UnProcessed* we simply remove it from the set. (If the set becomes empty, then we know that the system has become feasible.) However, the deletion of a constraint in the constraint graph can no longer be processed trivially: it may cause the infeasible system to become feasible. So, we remove the constraint from the constraint

```
function DeleteConstraint (<V,E, $length$>, UnProcessed, $v - u \leq c$, $D$)
declare
   <V,E, $length$>:  a feasible constraint graph
   $D$ :  a feasible solution for <V,E, $length$>
   UnProcessed:  a set of constraints
   $v - u \leq c$:  a constraint to be deleted
begin
   if ($v - u \leq c \in$ UnProcessed) then
      remove $v - u \leq c$ from UnProcessed
   else
      remove $v - u \leq c$ from E
      while (UnProcessed $\neq \phi$) do
         select a constraint $x - y \leq b$ from UnProcessed
         if ( AddToFeasible (<V,E, $length$>, $x - y \leq b$, $D$) then
            remove $x - y \leq b$ from UnProcessed
         else
            exit while loop
         fi
      od
   fi
end
```

Figure 3: Algorithm to update the constraint system after the deletion of a constrain.

graph. Then, we start processing the constraints in the set *UnProcessed* one by one, as if each had been just added, using the algorithm outlined in Section 5. While processing any one of these constraints, we may discover that the system continues to be infeasible, in which case we can stop. If we are able to process all constraints in *UnProcessed* and satisfy them, we know the new system in feasible, and we have a solution to the system.

Let us now consider the complexity of these algorithms. Note that when the constraint system is infeasible, all the work is done during the deletion of constraints, and the addition of constraints is trivially processed in O(1) time. In this case, it is more convenient to use amortized analysis [23] to measure the complexity of the update algorithm. It can be easily verified that the amortized complexity of processing the addition or deletion of a constraint (in an infeasible constraint system) is $O(m + n \log n)$.

7 Previous Work

Constraint Systems

Various types of constraint systems have been widely studied. Pratt[17] showed that a system of difference constraints could be represented by a weighted directed graph such that a system is feasible iff there exists no negative weight cycle in the graph. He also gave an $O(n^3)$ algorithm for solving systems of difference constraints, which uses the shortest-path algorithm. (Shortest paths can also be computed in $O(mn)$ time.)

Shostak [21] generalized Pratt's ideas to systems of two-variable linear constraints (constraints of the form $ax + by \leq c$, where a, b, and c are real constants and x and y are variables). He showed how such systems could be represented by a constraint graph such that a system is feasible iff the graph contains no cycle of a special kind. His algorithms for testing for feasibility, however, have an exponential worst-case behavior. Aspvall and Shiloach [2] improved Shostak's algorithm into a polynomial time algorithm. The most efficient algorithm currently known for this problem is an $O(mn^2 \log m)$ algorithm due to Hochbaum and Naor [9].

Le Pape [16] shows how one can deal with difference constraints over any totally ordered Abelian Group.

Jaffar et al. [11] considered the problem of two variable constraints of the form, $ax + by \leq c$, where $a, b \in \{-1, 0, 1\}$. They present an algorithm for computing a feasible solution to a system of two variable constraints that processes the constraints one by one. The algorithm takes $O(n^2)$ time per constraint, where n is the number of variables, and takes totally $O(m^3)$ time[2], where m is the number of constraints. Their algorithm can be directly used to update the solution in $O(n^2)$ time when a new constraint is added.

Incremental Algorithms

As explained earlier, Pratt has shown how the problem of computing a feasible solution to a system of difference constraints can be reduced to that of solving the single-source shortest-path (SSoSP) problem on a weighted graph (Theorem 2). Consequently, an incremental algorithm for maintaining the SSoSP solution can be used to maintain the solution of a system of difference constraints.

Ramalingam and Reps [20, 18] present an incremental algorithm, the RR algorithm, for updating the solution to the SSoSP problem in a graph in the absence of cycles of length zero. This algorithm was our starting point. However, the realization that maintaining the SSoSP solution itself was unnecessary for our problem led to the modified algorithm presented in this paper. The primary differences between our algorithm and the RR algorithm are: (1) Unlike the RR algorithm, our algorithm processes the deletion of an edge (constraint) in constant time since the solution is not updated. (2) Because the

[2]We believe the complexity of the algorithm can be more precisely described as being $O(mn^2)$.

solution is not updated when an edge is deleted, there is no guarantee that the feasible solution maintained is the shortest-path solution. The RR algorithm exploits the invariant that the current solution is the shortest-path solution to update the solution correctly after the insertion of an edge. Our primary contribution here has been establishing (in Theorems 4 and 5) that *even if this invariant is not maintained, the RR algorithm can still be used to correctly update the feasible solution for our problem.* (3) Relaxing the requirement that the SSoSP solution be maintained also means that cycles of length zero are no longer an issue in our problem, unlike in the case of the RR algorithm.

Ramalingam and Reps also measure the complexity of their incremental algorithm in terms of a parameter $\|\delta\|$ that measures the "size of the change in the input and output" and describe their incremental algorithm as being a *bounded* incremental algorithm since its complexity is bounded by some function of $\|\delta\|$. The parameter $\|\Delta\|$ we use is similar to, but not the same as, the parameter $\|\delta\|$, and the algorithm presented in this paper is not a bounded incremental algorithm. Unlike the shortest-path problem, a system of difference equations does not have a unique solution. For such problems (that have multiple solutions), a generalized parameter $\|\delta\|$ has been defined in terms of the smallest change to the current solution that produces a correct solution to a modified input instance [1]. The example in Figure 4 shows how the change to the solution produced by our algorithm may be larger than the minimal change required to produce a feasible solution and the example can be adapted to show that the algorithm presented in this paper is unbounded.

8 Extensions

Our algorithm does not necessarily identify the smallest change necessary to the current solution to produce a feasible solution. One reason for this is that our algorithm attempts to "propagate" the effects of the added constraint forward along the edges of the constraint graph. It would be worth exploring the possibility of "propagating" the effects of the added constraint along both directions of the edges (in a fashion similar to the one used by Alpern *et al.* in their incremental algorithm for updating a priority ordering [1]).

It would be worth exploring the possibility of adapting the algorithm presented in this paper or other "change-propagation" algorithms (for example, see Chapter 7 of [18]) to solve more general systems of constraints. (See, for instance, [19] for a generalization of the shortest-path problem and an incremental algorithm for the generalized problem.)

Systems of difference constraints are also useful in modelling scheduling problems. Scheduling problems, however, also have additional constraints involving certain *resources.* It would be interesting to see such additional resource constraints can be handled in the context of incremental updating.

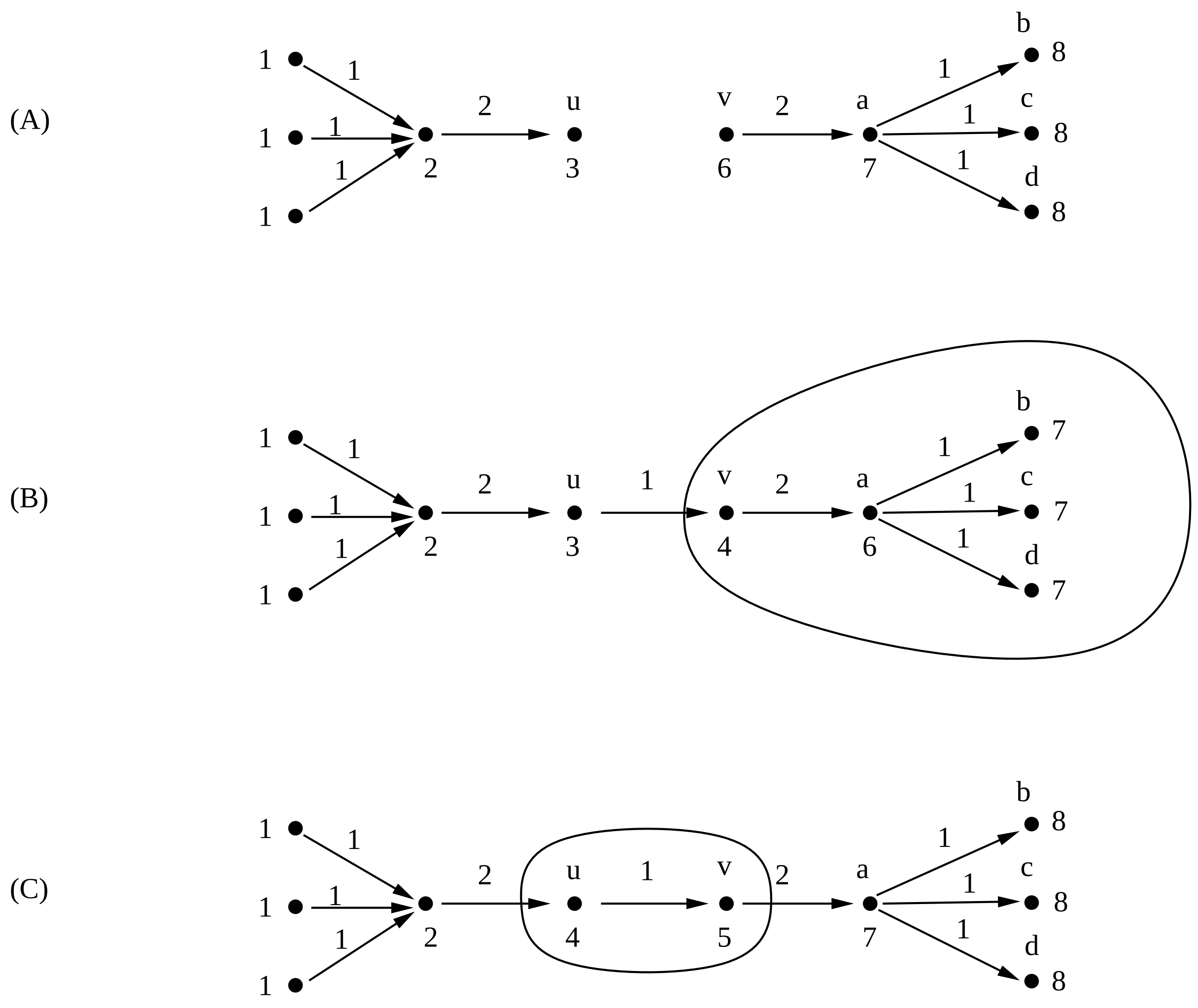

Figure 4: (A) A constraint graph and a feasible solution. (B) The graph after the addition of the constraint $v - u \leq 1$, and the updated solution computed by our algorithm. The values of variables v, a, b, c, and d are modified. (C) An alternative solution that involves changing the values of fewer variables (only u and v).

Acknowledgements

We would like to thank Tom Reps for his detailed comments which helped improve this paper, Jean-Louis Lassez and Dan Yellin for the useful discussions we had with them, and, finally, Dan Yellin again for bringing the first two authors together.

References

[1] B. Alpern, R. Hoover, B.K. Rosen, P.F. Sweeney, and F.K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, 1990.

[2] B. Aspvall and Y. Shiloach. A polynomial algorithm for solving systems of linear inequalities with two variables per inequalities. *SIAM Journal on Computing*, 9(4):827–844, 1980.

[3] M. C. Buchanan and P. T. Zellweger. Automatically generating consistent schedules for multimedia documents. 1:55–67, 1993.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[5] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint network. *Artificial Intelligence*, 49:61–95, 1991.

[6] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[7] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM*, 19:248–264, 1972.

[8] M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science Volume II*, pages 719–725. IEEE Computer Society, 1990.

[9] D. S. Hochbaum and Joseph Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, 1994.

[10] S. Hong and R. Gerber. Compiling real-time programs into schedulable code. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*, pages 166–176, 1993.

[11] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and Roland H. C. Yap. Beyond finite domains. In *Proceedings of Workshop on Principles and Practice of Constraint Programming*, pages 86–94, 1994.

[12] M. Kim and J. Song. Multimedia documents with elastic time. In *To appear in ACM Multimedia Conference'95*, 1995.

[13] I. Lee and V. Gehlot. Language constructs for real time programming. In *Proceedings of the IEEE Real Time Systems Symposium*, pages 57–66, 1985.

[14] Itay Meiri. Combining qualitative and quantitative constraints in temporal reasoning. Technical report, UCLA, 1991.

[15] V. Nirkhe, S. Tripathi, and A. Agrawala. Language support for the maruti real-time systems. In *Proceedings of IEEE Real Time Systems Symposium*, 1990.

[16] Le Pape. *Des Systemes d'Ordonnancement Flexibles et Opportunistes.* PhD thesis, Universite de Paris-Sud, 1988.

[17] V. R. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, September 1977.

[18] G. Ramalingam. *Bounded Incremental Computation.* PhD thesis, University of Wisconsin, Madison, WI, August 1993. Revised version to be published by Springer-Verlag in the *Lecture Notes in Computer Science* series.

[19] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. To appear in *Journal of Algorithms.*

[20] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. To appear in *Theoretical Computer Science A*, 162, July 1996.

[21] R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28:769–779, 1981.

[22] R.E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[23] R.E. Tarjan. Amortized computational complexity. *SIAM J. of Comput.*, 6(2):306–318, April 1985.

[24] W. Wolfe, S. Davidson, and I. Lee. Rtc: Language support for real-time concurrency. In *Proceedings of IEEE Real Time Systems Symposium*, pages 43–52, 1991.