Algorithmica (1999) 23: 261–275

Algorithmica

Solving Systems of Difference Constraints Incrementally

G. Ramalingam,[1] J. Song,[1] L. Joskowicz,[2] and R. E. Miller[3]

Abstract. Difference constraints systems consisting of inequalities of the form $x_i - x_j \leq b_{i,j}$ occur in many applications, most notably those involving temporal reasoning. Often, it is necessary to maintain a solution to such a system as constraints are added, modified, and deleted. Existing algorithms handle modifications by solving the resulting system anew each time, which is inefficient. 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 runs in $\Theta(mn)$ time, where n is the number of variables and m is the number of constraints.

This paper presents a new efficient incremental algorithm for maintaining a solution to a system of difference constraints. As constraints are added, modified, or deleted, the algorithm determines if the new system is feasible and updates its solution. When the system becomes infeasible, the algorithm continues to process changes until it becomes feasible again, at which point a feasible solution will be produced. The algorithm processes the addition of a constraint in time $O(m + n \log n)$ and the removal of a constraint in constant time when the original system is feasible. More precisely, additions are processed in time $O(\|\Delta\| + |\Delta| \log |\Delta|)$, where $|\Delta|$ is the number of variables whose values are changed to compute the new feasible solution, and $\|\Delta\|$ is the number of constraints involving the variables whose values are changed. When the original system is infeasible, the algorithm processes any change in $O(m + n \log n)$ *amortized* time. The new algorithm can also be used to check for the existence of negative cycles in dynamic graphs.

Key Words. Difference constraints, Incremental algorithm, Linear constraints, Shortest-path problem, Dynamic negative cycle.

1. Introduction.

A system of difference constraints is 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 occur in many applications involving temporal reasoning. In AI, Dechter et al. [5], [16] formulate a unifying temporal reasoning framework, called a temporal constraint network, based on difference constraints. Many real-time programming languages [12], [17], [15], [27] provide constructs that allow the specification of temporal relations as difference constraints. Multimedia applications also use difference constraints to specify temporal behavior [14], [3], [24]. For example, the play duration and relative ordering of multimedia objects such as audio or video segments are expressed as difference constraints relating the segment's time duration and minimum and maximum bounds.

Often it is necessary to maintain a solution to a difference constraints system as constraints are added, modified, and deleted. For example, interactive multimedia systems let users create difference constraint systems by adding and deleting constraints. As the system evolves, it is necessary to check for the feasibility of the new system of

[1] IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA.
[2] Institute of Computer Science, The Hebrew University, Jerusalem 91904, Israel.
[3] Department of Computer Science, University of Maryland, College Park, MD 20742, USA.

Received September 25, 1997; revised November 16, 1997. Communicated by F. P. Preparata.

constraints and report inconsistencies, if any, and to provide a solution, if the system is feasible. Existing algorithms handle modifications by solving the resulting system anew each time, which is inefficient. The best known algorithm to determine if a system of difference constraints is feasible and to compute a solution runs in $\Theta(mn)$ time, where n is the number of variables and m is the number of constraints.

This paper presents a new efficient incremental algorithm for testing the feasibility of a system of difference constraints and maintaining a solution to it. As constraints are added, modified, or deleted, the algorithm determines if the new system is feasible and updates its solution, providing the user with immediate feedback after each operation. If the system becomes infeasible, the algorithm will maintain the information and continue to process changes until it becomes feasible again, at which point a feasible solution will be produced. The algorithm processes the addition of a constraint in time $O(m + n \log n)$ and the removal of a constraint in constant time when the original system is feasible. More precisely, additions are processed in time $O(\|\Delta\| + |\Delta| \log|\Delta|)$, where $|\Delta|$ is the number of variables whose values are changed to compute the new feasible solution, and $\|\Delta\|$ is the number of constraints involving the variables whose values are changed. When the original system is infeasible, the algorithm processes any change in $O(m + n \log n)$ *amortized* time.

The rest of the paper is organized as follows. In Section 2 we briefly discuss previous work in this area. In Section 3 we present background material concerning systems of difference constraints. In Section 4 we define the problem addressed in this paper. In Section 5 we present a simple algorithm for the problem, which will help motivate an improved algorithm presented in Section 6. We assume in Sections 5 and 6 that the original system is feasible. In Section 7 we show how to handle infeasible systems. In Section 8 we compare our work with related work. In Section 9 we discuss possible future work.

2. Previous Work. Various types of constraint systems have been widely studied. Pratt [19] showed that a system of difference constraints can 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 [23] 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 [11]. Pape [18] shows how one can deal with difference constraints over totally ordered Abelian Groups. Jaffar et al. [13] 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,[4] 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.

As explained above, 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 (see Theorem 2 in Section 3). Consequently, an incremental algorithm for maintaining the SSoSP solution can be used to maintain the solution of a system of difference constraints. Several incremental algorithms [20], [22], [9], [10] have been developed for the SSoSP problem.

The starting point for our work was the algorithm presented by Ramalingam and Reps [22], [20] for updating the solution to the SSoSP problem in a graph in the absence of cycles of length zero. However, maintaining the SSoSP solution, while sufficient, is *unnecessary* for maintaining a solution to the system of difference constraints. This observation leads to the more efficient algorithm presented in this paper.

3. Preliminaries. A system of difference constraints $\langle V, C\rangle$ 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 $x_i, x_j \in V$ and $b_{i,j}$ is a constant. A feasible solution for a system of difference constraints is an assignment of real values to the variables (a function from 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 $G = \langle V, E, \mathit{length}\rangle$ consists of a set of vertices V, a set of edges E, and a function *length* from E to reals. We denote an edge from vertex u to vertex v by $u \to v$. We denote the length of a shortest path from u to v in a weighted graph G by $\mathit{dist}_G(u, v)$.

Pratt [19] has shown how a system of difference constraints $\langle V, C\rangle$ can be represented by a directed, weighted graph $G = \langle V, E, \mathit{length}\rangle$ whose vertices correspond to variables, and whose edges correspond to constraints.

DEFINITION 1 (Constraint Graph). The constraint graph of a system of difference constraints $\langle V, C\rangle$ is a directed, weighted graph $G = \langle V, E, \mathit{length}\rangle$ where

$$E = \{x_j \to x_i \mid x_i - x_j \leq a_{i,j} \in C\},$$

$$\mathit{length}(x_j \to x_i) = a_{i,j} \qquad \text{iff} \quad x_i - x_j \leq a_{i,j} \in C.$$

Thus, given a constraint graph, $\langle V, E, \mathit{length}\rangle$, the corresponding constraint system is $\langle V, \{v - u \leq \mathit{length}(u \to v) \mid u \to v \in E\}\rangle$.

Without loss of generality, we 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: 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

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

equally well to such multigraphs. Alternatively, $length(x_j \rightarrow x_i)$ may be defined to be $\min\{c \mid x_i - x_j \leq c \in C\}$. For 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 the purpose of computing a feasible solution:

DEFINITION 2 (Augmented Constraint Graph). The augmented constraint graph of a system of difference constraints $\langle V, C\rangle$ is a directed, weighted graph $G' = \langle V', E', length'\rangle$ where

$$V' = V \cup \{src\} \qquad \text{where} \quad src \notin V,$$

$$E' = \{\, x_j \rightarrow x_i \mid x_i - x_j \leq a_{i,j} \in C \,\} \cup \{src \rightarrow x_i | x_i \in V\},$$

$$length'(x_j \rightarrow x_i) = a_{i,j} \qquad \text{if} \quad x_i - x_j \leq a_{i,j} \in C,$$

$$length'(src \rightarrow x_i) = 0 \qquad \text{for} \quad x_i \in V.$$

THEOREM 1 [19] (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 [19]. *Let G be the augmented constraint graph of a consistent system of constraints $\langle V, C\rangle$. Then D is a feasible solution for $\langle V, C\rangle$, where*

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

4. 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 assume that the original system of constraints is 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 cannot 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: we 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 handled similarly.) 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 5 implements this idea. However, as we see in Section 6, this is more work than is necessary—it may often be easier to compute a feasible solution that is *not* the shortest-paths solution.

5. 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 the improved algorithm described in the next section. The algorithm determines if the addition of the new constraint makes the system infeasible. If it does not, it computes a feasible solution to the new set of constraints.

Checking 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. We 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 $dist_G(v, u) + c < 0$, where $dist_G(v, u)$ is the length of the shortest path from v to u in the original graph G. Hence, the problem reduces to computing $dist_G(v, u)$.

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. We can improve this bound by using the feasible solution for the original set of constraints by adapting Edmonds and Karp's technique for transforming the length of every edge to a nonnegative real without changing the graph's shortest paths [7], [25].

THEOREM 3 [7]. *Let $G = \langle V, E, length \rangle$ 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: $G_f = \langle V, E, length_f \rangle$, 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 scaling 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 nonnegative since D is a feasible solution for C. (Recall that the edge $x \rightarrow y$ corresponds to the constraint $y - x \leq length(x \rightarrow y)$, which can rewritten as $x + length(x \rightarrow y) - y \geq 0$.)

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 nonnegative. This enables us to use Dijkstra's algorithm to compute the shortest paths, and, hence, a feasible solution in $O(m + n \log n)$ time.

Several issues need to be addressed. First, note that we are now computing shortest paths in the augmented constraint graph. For the transformed edge lengths to be nonnegative 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 nonnegative in G_D for every edge that was in the original graph, this need not hold for the newly inserted edge $u \to v$. However, this is not a problem. Shortest path 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:

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

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

6. 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 nonnegative, and after that computes shortest-path lengths from scratch. We now show that changing the right-hand side of (1) lets us use the previous feasible solution more effectively in computing a new feasible solution.

Let $\langle V, C\rangle$ denote the original system of constraints, and let D be a feasible solution for $\langle V, C\rangle$. Let C' denote the new set of constraints $C \cup \{v - u \leq c\}$. Define D' as follows:[5]

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

[5] 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 (1) is equal to the right-hand side of (2),

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

THEOREM 4. *$\langle V, C' \rangle$ is feasible iff u is not affected.*

PROOF. We have $D'(u) = \min(D(u), D(u) + \mathit{length}(u \to v) + \mathit{dist}_G(v, u))$. Thus u is affected iff $D'(u) \neq D(u)$ iff $D(u) > D(u) + \mathit{length}(u \to v) + \mathit{dist}_G(v, u)$ iff $\mathit{length}(u \to v) + \mathit{dist}_G(v, u) < 0$ iff the new constraint graph has a negative cycle iff $\langle V, C' \rangle$ is infeasible. □

THEOREM 5. *If $\langle V, C' \rangle$ is feasible, then D' is a feasible solution for $\langle V, C' \rangle$.*

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

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

$$\begin{aligned} D'(y) &\leq D(y) && \text{(from the definition of } D') \\ &\leq D(x) + \mathit{length}(x \to y) && \text{(since } D \text{ is feasible)} \\ &\leq D'(x) + \mathit{length}(x \to y) && \text{(by assumption).} \end{aligned}$$

Case 2: $D'(x) = D(u) + \mathit{length}(u \to v) + \mathit{dist}_G(v, x)$.

$$\begin{aligned} D'(y) &\leq D(u) + \mathit{length}(u \to v) + \mathit{dist}_G(v, y) && \text{(from the definition of } D') \\ &\leq D(u) + \mathit{length}(u \to v) + \mathit{dist}_G(v, x) \\ &\quad + \mathit{length}(x \to y) && \text{(}\mathit{dist}\text{ property)} \\ &\leq D'(x) + \mathit{length}(x \to y). \end{aligned}$$

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

$$\begin{aligned} D'(v) &= \min(D(v), D(u) + \mathit{length}(u \to v) + \mathit{dist}_G(v, v)) \\ &= \min(D(v), D(u) + \mathit{length}(u \to v)) \\ &\leq D(u) + \mathit{length}(u \to v) \\ &\leq D'(u) + \mathit{length}(u \to v). \end{aligned}$$

Hence, the new constraint $v \leq u + \mathit{length}(u \to v)$ 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 ($\mathit{dist}_{G'}(\mathit{src}, x)$) computed by our original algorithm. We now show how D' can be computed efficiently.

---

which is easier to compute since it requires the computation of only $\mathit{dist}_G(v, x)$. 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 (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.

Computing D'

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

PROOF.

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

Hence y is also unaffected. □

We say that an edge $x \to y$ is affected iff vertex x is affected. Let H denote the subgraph of G consisting only of affected edges. Thus, $H = \langle V, E_a, \mathit{length}|E_a \rangle$, where E_a is the set of affected edges and $\mathit{length}|E_a$ denotes the restriction of *length* to E_a.

THEOREM 7.

$$D'(x) = \min(D(x), D(u) + \mathit{length}(u \to v) + \mathit{dist}_H(v, x)).$$

PROOF. Obviously, $\mathit{dist}_H(v, x) \geq \mathit{dist}_G(v, x)$. Theorem 6 implies that $\mathit{dist}_H(v, x) = \mathit{dist}_G(v, x)$ for all affected vertices x. The result follows. □

The algorithm in Figure 1 shows how we can utilize the results presented so far to identify efficiently if the new system is feasible and, if it is, to compute a new feasible solution 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 $\mathit{InsertIntoHeap}(H, i, k)$ inserts an item i into heap H with a key k. The operation $\mathit{FindAndDeleteMin}(H)$ 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 $\mathit{AdjustHeap}(H, i, k)$ 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 $\mathit{KeyOf}(H, i)$ returns the key of item i in heap H if i is in H and ∞ otherwise.

Lines [4]–[7] and [15]–[20] 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).

We now analyze the complexity of the algorithm in Figure 1. It is straightforward to

```
function AddToFeasible (<V,E, length>, v − u ≤ c, D)
parameters
      <V,E, length>:  a feasible constraint graph
      D : a feasible solution for <V,E, length>
      v − u ≤ 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 → v to E
[2]   length(u → v) := c
[3]   D′ := D
[4]   PriorityQueue := φ
[5]   InsertHeap(PriorityQueue, v, 0)
[6]   while PriorityQueue != φ do
[7]      (x, dist_x) := FindAndDeleteMin(PriorityQueue)
[8]      if D(u) + length(u → v) + (D(x) + dist_x − D(v)) < D(x) then
[9]         if (x = u) then
[10]           /* Infeasible system:  reject new constraint */
[11]           remove edge u → v from E
[12]           return false
[13]        else
[14]           D′(x) := D(u) + length(u → v) + (D(x) + dist_x − D(v))
[15]           for every vertex y in Succ(x) do
[16]              scaledPathLength := dist_x + (D(x) + length(x → 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
```

Fig. 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 $\langle V, E, \mathit{length}\rangle$ would introduce infeasibility. Otherwise, it adds the new constraint to the system, updates the solution D to the system of constraints, and returns `true`.

implement the maps D and D' so that lines [3] and [25] can be implemented in constant time. 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

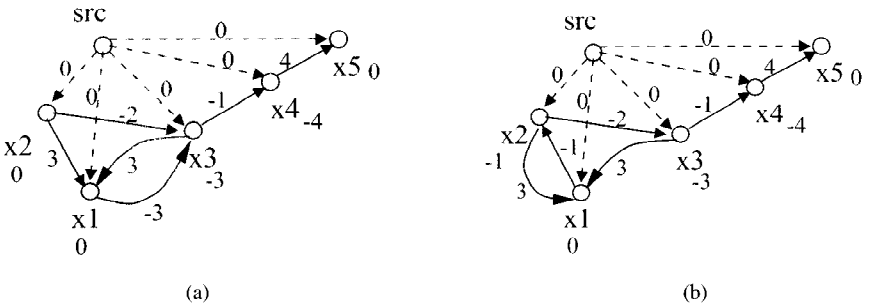

Fig. 2. Example: augmented constraint graphs and solutions. (a) Graph for the original system. (b) Graph after deletion of $x_3 - x_1 \leq -3$ and addition of $x_2 - x_1 \leq -1$.

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 $O(\log|\Delta|)$ 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 $O(\|\Delta\| + |\Delta|\log|\Delta|/\log\log|\Delta|)$.

6.1. *Example.* Consider the following system of constraints: $\{x_1 - x_2 \leq 3, x_3 - x_2 \leq -2, x_1 - x_3 \leq 3, x_3 - x_1 \leq -3, x_4 - x_3 \leq -1, x_5 - x_4 \leq 4\}$. The augmented constraint graph for the system is shown in Figure 2(a). The graph contains no negative cycles and the system is feasible. Assume that the current feasible solution is $(x_1, x_2, x_3, x_4, x_5) = (0, 0, -3, -4, 0)$, which is the shortest-path solution.

Now, we assume that the constraint $x_3 - x_1 \leq -3$ is deleted. The system remains feasible and our incremental algorithm does not modify the current feasible solution. At this point, the current feasible solution is no longer the shortest-path solution.

Now consider the addition of a new constraint $x_2 - x_1 \leq -1$ to the system. The algorithm initially adds x_2 to `PriorityQueue` with priority 0. Inside the loop, x_2 is determined to be affected, and its value is updated to -1. The successors of x_2, namely, x_1 and x_3, are then added to `PriorityQueue` with priorities 3 and 1, respectively (lines [15]–[20]). In the subsequent iterations of the loop, it is determined that neither x_3 nor x_1 is affected, and the algorithm halts with the new feasible solution $(x_1, x_2, x_3, x_4, x_5) = (0, -1, -3, -4, 0)$. (See Figure 2(b).) Observe that the algorithm does not even examine variables x_4 or x_5 or the constraints involving them. In a much larger system of constraints, the benefits of using the incremental algorithm can be even better.

However, let us see what happens if instead of adding the constraint $x_2 - x_1 \leq -1$ we add the constraint $x_2 - x_1 \leq -2$. The system will proceed as before initially. However, this time the system will discover that x_3 is affected, and then discover that x_1 is also affected. At this point, the system knows that the new system of constraints is infeasible

and halts (lines [10]–[12]). The addition of the new constraint introduces a negative cycle (x_1, x_2, x_3, x_1).

7. 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. Refer to the algorithms in Figures 3 and 4.

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, of the remaining constraints, called *UnProcessed*. When the system is feasible, *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; when *UnProcessed* becomes empty, we know that the system has become feasible. The deletion of a constraint from the constraint graph can no

```
function AddConstraint (<V,E, length>, UnProcessed, v − u ≤ 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 ≤ 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 ≤ c, D) then
         /* Adding new constraint will cause infeasibility */
         UnProcessed := { v − u ≤ c }
      fi
   else
      /* System is infeasible before addition of new constraint */
      Add v − u ≤ c to UnProcessed
   fi
end
```

Fig. 3. Algorithm to update the constraint system after the addition of a constraint.

```
function DeleteConstraint (<V,E, length>, UnProcessed, v − u ≤ 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 ≤ c:  a constraint to be deleted
begin
   if (v − u ≤ c ∈ UnProcessed) then
      remove v − u ≤ c from UnProcessed
   else
      remove v − u ≤ c from E
      while (UnProcessed ≠ φ) do
         select a constraint x − y ≤ b from UnProcessed
         if ( AddToFeasible (<V,E, length>, x − y ≤ b, D) then
            remove x − y ≤ b from UnProcessed
         else
            exit while loop
         fi
      od
   fi
end
```

Fig. 4. Algorithm to update the constraint system after the deletion of a constraint.

longer be processed trivially: it may cause the infeasible system to become feasible. We thus first remove the constraint from the constraint graph and then process the constraints in the set *UnProcessed* one by one, as if each had been just added, using the algorithm outlined in Section 6. 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 is feasible, and we have a solution to the system.

We 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 [26] 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 $O(m + n \log n)$.

8. Related Work. The starting point for the work described in this paper was the algorithm presented by Ramalingam and Reps [22], [20], the RR algorithm, for updating the solution to the SSoSP problem in a graph in the absence of cycles of length zero. 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. Our algorithm, unlike the RR algorithm, processes the deletion of an edge (constraint) in constant time since the solution is not updated.
2. Because the 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 is establishing, through Theorems 4 and 5, that *even if this invariant is not maintained, the RR algorithm can still be used to update the feasible solution correctly 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.

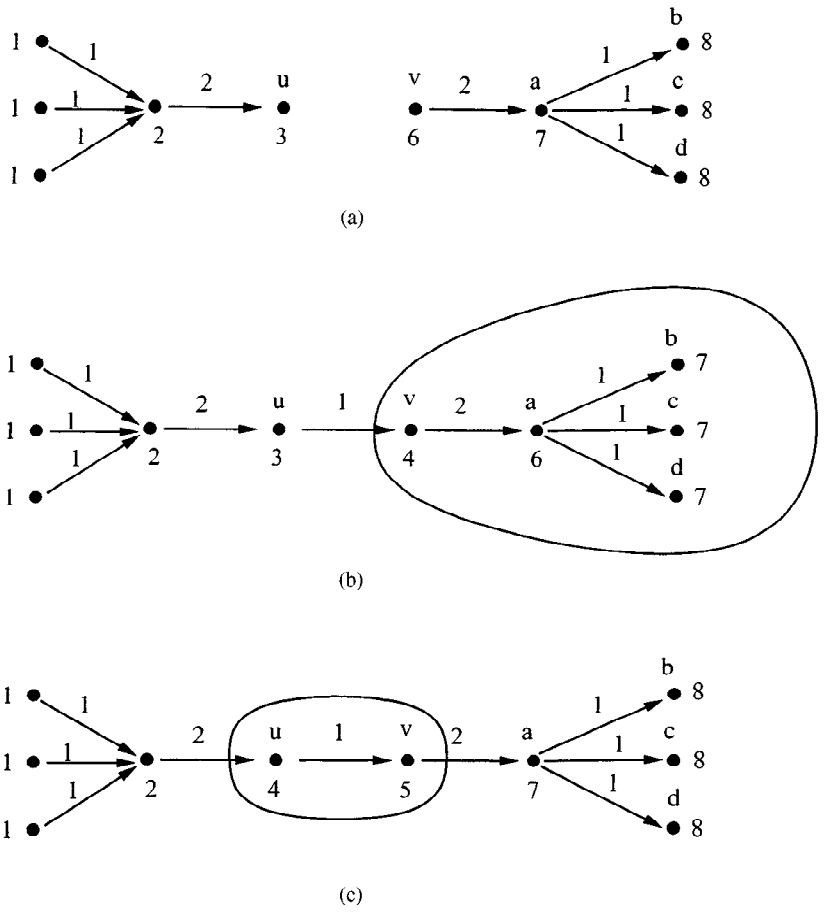

Fig. 5. (a) A constraint graph and a feasible solution. (b) The graph after the addition of the constraint $v - u \leq 1$, and the updated solution computed by our algorithm. The values of variables v, a, b, c, and d are modified. (c) An alternative solution that involves changing the values of fewer variables (only u and v).

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 constraints does not have a unique solution. For 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 5 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.

9. 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. Alternatively, we could propagate the effects of the added constraint along both directions of the edges, as in Alpern et al.'s incremental algorithm for updating a priority ordering [1]. Another interesting avenue is trying to adapt the algorithm presented in this paper or other "change-propagation" algorithms (for example, see Chapter 7 of [20]) to solve more general systems of constraints. See, for instance, [21] for a generalization of the shortest-path problem and an incremental algorithm for the generalized problem.

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

Acknowledgments. 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

[3] M. C. Buchanon and P. T. Zellweger. Automatically generating consistent schedules for multimedia documents. *Multimedia Systems Journal*, 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 computatio. *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 the 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* 31*st Annual Symposium on Foundations of Computer Science*, Volume II, pages 719–725. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[9] D. Frigoni, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for the single-source shortest path problem. In P. S. Thiagarajan, editor, *Proceedings of the Fourteenth FST&TCS Conference*, Lecture Notes in Computer Science, Volume 880, pages 113–124. Springer-Verlag, Berlin, 1994.

[10] D. Frigoni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problem. In *Proceedings of the Seventh Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 212–221, 1996.

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

[12] 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.

[13] J. Jaffar, M.J. Maher, P.J. Stuckey, and R.H.C. Yap. Beyond finite domains. In *Proceedings of Workshop on Principles and Practice of Constraint Programming*, pages 86–94, 1994.

[14] M. Kim and J. Song. Multimedia documents with elastic time. In *Proceedings of the ACM Multimedia Conference* '95, pages 143–154, 1995.

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

[16] I. Meiri. Combining qualitative and quantitative constraints in temporal reasoning. *Artificial Intelligence*, 87:343–385, 1996.

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

[18] L. Pape. Des Systemes d'Ordonnancement Flexibles et Opportunistes. Ph.D. thesis, Universite de Paris-Sud, 1988.

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

[20] G. Ramalingam. *Bounded Incremental Computation*. Volume 1089. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1996.

[21] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267–305, 1996.

[22] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

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

[24] J. Song, M. Kim, R. Miller, G. Ramalingam, and B. Yi. Interactive authoring of multimedia documents. In *Proceedings of the* 1996 *Symposium on Visual Languages*, pages 276–283, September 1996.

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

[26] R. E. Tarjan. Amortized computational complexity. *SIAM Journal of Computing*, 6(2):306–318, April 1985.

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