Algorithmica (2004) 39: 255–274
DOI: 10.1007/s00453-004-1081-6

Algorithmica

Solving Systems of Difference Constraints Incrementally with Bidirectional Search

Jianjun Zhou[1] and Martin Müller[1]

Abstract. We propose an incremental algorithm for the problem of maintaining systems of difference constraints. As a difference from the unidirectional approach of Ramalingam et al. [16], it employs bidirectional search, which is similar to that of Alpern et al. [1], and has a bounded runtime complexity in the worst case in terms of the size of changes. The major challenge is how to update the solution efficiently after the bidirectional search discovers a region that needs changes. Experimental results show that our approach is much faster in runtime and generates much smaller changes than the algorithm in [16]. We also perform an experimental study on the edge value heuristic [1] and results show that a simpler method may be faster in practice.

Key Words. Incremental algorithm, Difference constraints, Bidirectional search, Shortest-path algorithm, Computational complexity.

1. Introduction. A system of difference constraints (SDC) is a set of inequalities of the form $x_i - x_j \leq b_{ij}$ [16]. Problems involving difference constraints arise in various areas of computer science such as scheduling, planning, multimedia, and parallel computation [5], [16], [11]. A *simple temporal problem* defined in [5] represents constraints in the form $a_{ij} \leq x_j - x_i \leq b_{ij}$. Alternatively, such constraints can also be expressed as a pair of linear inequalities:

$$x_j - x_i \leq b_{ij},$$

$$x_i - x_j \leq -a_{ij}.$$

As discussed in [16], the problem of computing a solution to an SDC can be reduced to that of solving the single-source shortest-path (SSoSP) problem on a weighted graph. By using the Bellman–Ford algorithm, we can compute a solution to an SDC in $O(nm)$ time, where m is the number of constraints and n is the number of variables [4]. However, many applications involve adding, deleting, and modifying constraints in a difference constraints system. In such a case, incrementally maintaining a solution is often much cheaper than recomputing everything. For example, Ramalingam et al. give an algorithm processing the addition of a constraint in time $O(|C_\Delta| + |\Delta| \log |\Delta|)$, where $|\Delta|$ is the number of variables whose values are changed to compute the new solution, and $|C_\Delta|$ is the number of constraints involving the variables whose values are changed [16].

Because deleting a constraint can be done directly without violating any other constraint, and modifying a constraint can be trivially reduced to the problem of adding constraints, in this paper we only discuss adding constraints.

[1] 221 Athabasca Hall, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8. {jianjun,mmueller}@cs.ualberta.ca.

Received February 3, 2003; revised October 10, 2003. Communicated by H. N. Gabow.
Online publication February 25, 2004.

This paper is organized in the following manner: we discuss systems of difference constraints and previous work on this topic in Section 2. We describe our algorithm in Section 3. Thereafter we present the experimental results in Section 4. Finally, in Section 5 we discuss possible extensions.

2. Preliminaries and Related Work

2.1. *System of Difference Constraints* (*SDC*). An SDC [16] $\langle V, C\rangle$ consists of a set V of variables and a set C of linear inequalities of the form $x_i - x_j \leq b_{ij}$, where $x_i, x_j \in V$ and b_{ij} is a constant. Such a system can be represented by a constraint graph.

DEFINITION 1 (Constraint Graph [13], [16]). 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

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

$$\mathit{length}(x_j \to x_i) = a_{ij} \quad \Leftrightarrow \quad x_i - x_j \leq a_{ij} \in C.$$

For any given graph, let $x \xrightarrow{*} y$ denote a (possibly empty) path from x to y and let $x \xrightarrow{+} y$ denote a nonnull path from x to y. In a constraint graph, any nonnull path corresponds to a constraint between the endpoints of the path.

THEOREM 1. *Given a constraint graph $G = \langle V, E, \mathit{length}\rangle$, $\forall s, t \in V$, if there is a nonnull path $s \xrightarrow{+} t$ and $t - s > \mathit{length}(s \xrightarrow{+} t)$, then at least one edge in $s \xrightarrow{+} t$ violates the corresponding constraint.*

PROOF. ($\Rightarrow$) Suppose every edge on $s \xrightarrow{+} t$ satisfies the corresponding constraint. Let $x_i \to x_{i+1}$ be the ith edge in $s \xrightarrow{+} t$ ($i = 1, 2, \ldots$), then

$$\begin{aligned} x_2 - x_1 &\leq \mathit{length}(x_1 \to x_2);\\ x_3 - x_2 &\leq \mathit{length}(x_2 \to x_3);\\ &\vdots\\ x_n - x_{n-1} &\leq \mathit{length}(x_{n-1} \to x_n). \end{aligned}$$

By summing both sides, we get $x_n - x_1 \leq \mathit{length}(s \xrightarrow{+} t)$. Because $x_1 = s$ and $x_n = t$, $t - s \leq \mathit{length}(s \xrightarrow{+} t)$, a contradiction. □

DEFINITION 2 (Subpath). Let $s \xrightarrow{*} t$ be a path with a sequence

$$\langle v_0, v_1, v_2, \ldots, v_k\rangle$$

of vertices such that $s = v_0$, $t = v_k$. If $x \xrightarrow{*} y$ is path with the sequence

$$\langle v_i, v_{i+1}, \ldots, v_j\rangle$$

such that $x = v_i$, $y = v_j$, and $0 \le i \le j \le k$, then $x \xrightarrow{*} y$ is a subpath of $s \xrightarrow{*} t$, denoted by $x \xrightarrow{*} y \subseteq s \xrightarrow{*} t$.

THEOREM 2. *Given an edge $u \to v$ and two paths $s \xrightarrow{+} t$, $x \xrightarrow{+} y$ such that $u \to v \subseteq x \xrightarrow{+} y \subseteq s \xrightarrow{+} t$, if $u \to v$ is the only edge in $s \xrightarrow{+} t$ that violates the constraint and $y - x \le length(x \xrightarrow{+} y)$, then $t - s \le length(s \xrightarrow{+} t)$.*

PROOF. By summing up $t - y \le length(y \xrightarrow{*} t)$, $y - x \le length(x \xrightarrow{+} y)$, and $x - s \le length(s \xrightarrow{*} x)$, we get $t - s \le length(y \xrightarrow{*} t) + length(x \xrightarrow{+} y) + length(s \xrightarrow{*} x) = length(s \xrightarrow{+} t)$. □

An SDC is called feasible if there exists a solution to the system of inequalities. In order to compute a feasible solution, we can augment the graph with an extra source vertex and extra edges from this vertex to every vertex in the original graph.

DEFINITION 3 (Augmented Constraint Graph [16]). The augmented constraint graph of an SDC $\langle V, C\rangle$ is a directed, weighted graph $G' = \langle V', E', length'\rangle$ where

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

$$E' = \{x_j \to x_i \mid x_i - x_j \le a_{ij} \in C\} \cup \{src \to x_i \mid x_i \in V\},$$

$$length'(x_j \to x_i) = a_{ij} \qquad \text{if} \quad x_i - x_j \le a_{ij} \in C,$$

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

We have the following theorems.

THEOREM 3 [13], [16]. *A system of difference constraints is feasible if and only if there are no negative cycles in its corresponding constraint graph.*

THEOREM 4 [13], [16]. *Let G' be the augmented constraint graph of an SDC $\langle V, C\rangle$, and let $dist_{G'}(src, u)$ be the distance between the vertices src and u in G'. Then D is a feasible solution for $\langle V, C\rangle$, where $D(u) = dist_{G'}(src, u)$.*

By Theorem 4, the problem of generating a solution to the system is reduced to solving an SSoSP problem, and we can apply Dijkstra's algorithm. In general, Dijkstra's algorithm requires the length of the edges to be nonnegative. In order to handle negative length edges, Edmonds and Karp proposed the technique of scaling the length of every edge to be nonnegative, without changing the shortest paths of the graph.

THEOREM 5 [7], [16]. *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*

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

A path 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

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

THEOREM 6. *Let $G = \langle V, E, length\rangle$ be a constraint graph and let D be a real-valued function on V. For all nonnull paths $s \xrightarrow{+} t$ in G, $D(t) - D(s) > length(s \xrightarrow{+} t)$ if and only if*

$$length_D(s \xrightarrow{+} t) = D(s) + length(s \xrightarrow{+} t) - D(t) < 0.$$

PROOF. Let $s, x_1, x_2, \ldots, t$ be the nodes along $s \xrightarrow{+} t$. Then

$$\begin{aligned} length_D(s \xrightarrow{+} t) &= length_D(s \to x_1) + length_D(x_1 \to x_2) + \cdots \\ &= D(s) + length(s \to x_1) - D(x_1) + D(x_1) \\ &\quad + length(x_1 \to x_2) - x_2 + \cdots \\ &= D(s) + length(s \xrightarrow{+} t) - D(t), \end{aligned}$$

$D(t) - D(s) > length(s \xrightarrow{+} t) \quad \Leftrightarrow \quad D(s) + length(s \xrightarrow{+} t) - D(t) < 0 \quad \Leftrightarrow$ $length_D(s \xrightarrow{+} t) < 0$. □

COROLLARY 7. *Let C be a cycle in G. Then $length(C) < 0$ if and only if $length_D(C) < 0$.*

Hence the problem of finding paths violating constraints in a constraint graph G can be reduced to finding paths of negative length in G_D.

2.2. *Measures of Runtime Complexity in Incremental Algorithms.* Incremental problem solving can be explained as follows: given an original problem, its valid solution, and some changes to the original problem, change the original solution to obtain a solution to the modified problem. Incremental algorithms focus on making changes to the existing solution, rather than computing a whole solution from scratch. The methods they use and the measurement of such methods can be completely different from nonincremental algorithms.

The traditional way of evaluating the computational complexity of an algorithm is to express the cost of the computation as a function of the size of the input and use asymptotic worst-case analysis. However, for incremental algorithms, as pointed out in [1] and [15], such analysis is not as informative as using the size of the changes in the input and output. For example, for some problems no incremental algorithms exist that have asymptotically better worst-case runtime than performing the computation from scratch [2], [15].

A further issue, discussed by Alpern et al., is that the number of changed nodes is not enough to measure the actual cost of an incremental algorithm, because the number of edges incident with nodes in the changes can significantly influence the speed of processing. They propose a better measure, the *extended size* [1].

DEFINITION 4 (Extended Size [1]). Given a graph $G = \langle V, E \rangle$ and a set of nodes $\mathcal{K} \subseteq V$, the extended size of $\mathcal{K}$, denoted as $\|\mathcal{K}\|$, is defined by

$$\|\mathcal{K}\| = |\mathcal{K}| + |Touch(\mathcal{K})|,$$

where $Touch(\mathcal{K})$ is the set of edges in E that are incident with nodes in $\mathcal{K}$.

In this paper we follow the approach of [1], [15], and [16] and use $\|\mathcal{K}\|$ to analyze the performance of our algorithm.

2.2.1. *Bounded Incremental Algorithms*. A standard approach in algorithm analysis is to compare the performance of a concrete algorithm with that of the hypothetical optimal algorithm. For incremental algorithms, this knowledge is especially important, as such algorithms are frequently applied in online updating systems that need to respond to input changes in time. An incremental algorithm for a particular problem is said to be *bounded* if, for any changes to the input of the problem, its runtime R depends on the optimal changes of the output only. In another word, there exists a function f, such that

$$R \leq f(\min(\text{the size of possible changes to the output})).$$

If a problem has bounded incremental algorithms, it is said to be bounded. Otherwise, it is unbounded. An example of an unbounded problem is the problem of incrementally maintaining an SSoSP solution. The unboundedness is due to the issue of zero length cycles [15], [16]. However, maintaining an SDC incrementally does not have this problem. In contrast to the algorithm in [16], which is not bounded, we prove in Section 3 that our new algorithm is bounded.

2.3. *Unidirectional and Bidirectional Search*. Unidirectional graph search proceeds from a start node s towards some goal node t, while bidirectional search proceeds both in the forward direction from s and in the backward direction from t [8], [12], [17]. In this paper we use the term bidirectional search more loosely, and refer to any search that proceeds in two directions as bidirectional search.

The basic idea of [16] is to apply Dijkstra's algorithm, a typical unidirectional search, with the scaling technique to update the shortest path values when the inserted edge violates the constraints. If deletions are allowed, the solution is not guaranteed to be an SSoSP solution [16]. The process begins from the destination of the inserted edge, v. Only v and the nodes reachable from v will be affected. While searching forward, the algorithm also performs negative cycle detection since negative cycles will lead to an infeasible system. Whenever the search frontier meets the source of the inserted edge, u, a cycle is found and its length is checked. Because the original system is feasible, if the insertion generates any negative cycles, they must include the inserted edge [16]. Thus by checking v and its descendants, the algorithm returns an updated solution if the system is still feasible, or otherwise stops when a negative cycle is detected.

The major drawback of this approach is that the algorithm is not bounded, because it is possible that the algorithm updates a large number of vertices in the forward direction, while the smallest solution would only require a few changes in the backward direction [16].

An alternative approach is to use bidirectional search, which may work better as demonstrated by Alpern et al. in their incremental algorithm for maintaining topological ordering [1]. After an edge is inserted into a graph, this algorithm performs two best-first searches, from the source node and the target node of the inserted edge, respectively, to form two search frontiers. Comparing properties of the search frontier nodes yields the criteria for when to stop the search. Furthermore, that algorithm uses an *edge value heuristic* to decide which of the two search frontiers to expand at each step. This leads to a runtime complexity that is polynomially bounded in terms of the size of changes, and this size is guaranteed to be no greater than a factor of 3 times the optimum.

Ramalingam et al. [16] propose to investigate whether an approach similar to [1] can be used to solve the problem of difference constraints incrementally. Our answer is affirmative. Theorems 1 and 2 show that constraints on the paths between the two search frontiers can be used to decide the scope of the constraints violation caused by inserted edges. The major challenge then is how to use the information of the frontiers to update the solution efficiently.

In a classical nonincremental setting, bidirectional search has been shown to provide substantial computational savings in the two-node shortest-path problem [12], [6]. Although a bidirectional search has more overhead than a unidirectional one, it can be much faster when the changes only affect a small portion of the graph, and the branching factor, or the average degree, is large. Let the branching factor be b and let the distance between the start node s and the target node t be d. A unidirectional breadth-first search needs to visit $O(b^d)$ nodes while a bidirectional one only needs to visit $O(2b^{d/2})$ nodes which is significantly smaller when b and d are large.

3. Algorithm.

When using bidirectional search as in [1] for our problem, the major challenge is how to make efficient use of the frontier information to update the solution. Similar to [1], our algorithm also contains two parts, the Discovery Algorithm (DA) and the Reassignment Algorithm (RA). First, DA computes a cover, a set of nodes within which a solution to the modified problem can be found. Second, RA reassigns the solution values for the nodes in the cover.

3.1. *The Discovery Algorithm* (*DA*).

DA (Figure 1) has two purposes: if the constraint graph is still feasible after a constraint is added, it computes a cover, the set of nodes for which to apply changes. Otherwise, if the graph is not feasible, it detects a negative cycle. We apply Dijkstra's algorithm with the scaling technique in the forward and backward direction, performing two separate but related frontier searches. By Theorems 2 and 6, if all scaled shortest paths between the two frontiers have a nonnegative length, then no path outside the region between the forward and backward frontier is violating the corresponding constraint, so that the algorithm stops.

Similar to [1], we define a *cover* as a region in which to apply corresponding changes when a new edge is inserted.

DEFINITION 5 (Cover). Given a constraint graph $G = \langle V, E, length \rangle$, a set of nodes, $\mathcal{K}$, is a cover if, $\forall x, y \in V$,

$$x \xrightarrow{+} y \ \wedge \ y - x > length(x \xrightarrow{+} y) \quad \Rightarrow \quad x \in \mathcal{K} \vee y \in \mathcal{K}.$$

```
Input:
   G = ⟨V, E, length⟩: the constraint graph;
   D: a feasible solution for G;
   u → v: an edge to insert, which violates D;
Output:
   MarkedForwQ and MarkedBackQ: nodes marked in forward and backward direction;
   SSPE value;
   b,f: the frontier outputs when DA stops.

 1: Init()
 2:     for all x do
 3:         SSPE(v, x):=+∞; SSPE(x, u):=+∞;
 4:     SSPE(u, u):=0; SSPE(v, v):=0;
 5:     PriorityQueuePush(em ForwFron, v, 0); PriorityQueuePush(BackFron, u, 0);
 6:     MarkedForwQ := ∅; MarkedBackQ := ∅;
 7:
 8: PriorityQueuePush(frontier, node, priority)
 9:     edgeVal(node) := degree(node);
10:    Push(frontier, node, priority);
11:
12: Main()
13:     Init();
14:     while GetMin(ForwFron, f) ∧ GetMin(BackFron, b)
15:             ∧SSPE(b, u) + ScaledLength(u → v) + SSPE(v, f) < 0 do
16:         μ:= Min(NumEdges(b), NumEdges(f));
17:         NumEdges(b) := NumEdges(b) − μ; NumEdges(f) := NumEdges(f) − μ;
18:         if NumEdges(f) = 0 then
19:             DeleteMin(ForwFron); ExtendForw(f);
20:         if NumEdges(b) = 0 then
21:             DeleteMin(BackFron); ExtendBack(b)
22:     return true;
23:
24: ExtendForw(x)
25:     MarkedForwQ.append(x);
26:     for all y with x → y do
27:         NewVal = SSPE(v, x) + (D(x) + length(x → y) − D(y));
28:         if SSPE(v, y) > NewVal then
29:             if NewVal + SSPE(y, u) + ScaledLength(u → v) < 0 then
30:                 Algorithm exit with false; /* find a negative cycle */
31:             if SSPE(v, y) = +∞ then /* not visited forward before */
32:                 SSPE(v, y) := NewVal;
33:                 PriorityQueuePush(ForwFron, y, SSPE(v, y));
34:             else do
35:                 SSPE(v, y) := NewVal;
36:                 AdjustHeap(ForwFron, y, SSPE(v, y));
37:
38: ExtendBack(x)
39:     MarkedBackQ.append(x);
40:     for all y with y → x do
41:         NewVal = SSPE(x, u) + (D(y) + length(y → x) − D(x));
42:         if SSPE(y, u) > NewVal then
43:             if SSPE(v, y) + NewVal + ScaledLength(u → v) < 0 then
44:                 Algorithm exit with false;
45:             if SSPE(u, y) = +∞ then
46:                 SSPE(y, u) := NewVal;
47:                 PriorityQueuePush(BackFron, y, SSPE(y, u));
48:             else do
49:                 SSPE(y, u) := NewVal;
50:                 AdjustHeap(BackFron, y, SSPE(y, u));
```

Fig. 1. The Discovery Algorithm.

For each direction, a min-priority queue maintains the scaled shortest-path estimates of the frontier nodes. Scaled shortest-path estimates are defined by the corresponding concept in Dijkstra's algorithm [4]. Note that in the backward direction, we apply Dijkstra's algorithm in a reverse way, using u as the source.

DEFINITION 6 (Scaled Shortest-Path Estimate). Given a graph $G = \langle V, E, length \rangle$ and two nodes $s, t \in V$, let D be a real-valued function on V and let the shortest-path estimate from s to t be SPE, then the scaled shortest-path estimate, denoted as $SSPE(s, t)$, is defined as

$$SSPE(s, t) = D(s) + SPE(s, t) - D(t).$$

In our incremental algorithm, D is the previous solution. In every iteration, a node with minimal $SSPE$ value is extracted from each frontier. We call b and f the nodes for the backward frontier and forward frontier, respectively. These two nodes form the shortest path between the two frontiers. If its length is negative, then at least one of the nodes should be marked as part of the cover, and the frontier will be extended from this node. Otherwise, any scaled path between the two frontiers has a nonnegative length, and DA stops. Since choosing which of the two nodes to mark might influence the resulting size of the cover, we apply the edge value heuristic of [1] as follows: When a node is pushed to the priority queue, its edge value is initialized by its degree. Every time two nodes are extracted, the one with a smaller edge value is chosen to mark and both edge values are decreased by this smaller edge value. If the two edge values are equal, then both nodes are chosen. This heuristic gives our algorithm a bounded cover and bounded runtime complexity, as in [1].

THEOREM 8. *Given a constraint graph $G = \langle V, E, length \rangle$ with a feasible solution D and a new edge $u \to v$ to be added to G. If the system is still feasible,*

(a) *DA marks a cover $\mathcal{K}$;*
(b) *$\|\mathcal{K}\| \leq 3k$, where k is the minimum extended size for a cover;*
(c) *the worst-case running time is $O(k \log k)$.*

PROOF. (a) Suppose a nonnull path $s \xrightarrow{+} t$ violates the constraint such that $D(t) - D(s) > length(s \xrightarrow{+} t)$ but neither s nor t is in $\mathcal{K}$. As D is a feasible solution for G, $s \xrightarrow{+} t$ must contain $u \to v$. Let *ForwFron* and *BackFron* be the forward and backward frontier, respectively, when DA stops, then $s \xrightarrow{+} t \cap ForwFron \neq \emptyset$ and $s \xrightarrow{+} t \cap BackFron \neq \emptyset$; otherwise, at least one of s and t is marked. Let

$$i \in s \xrightarrow{+} t \cap ForwFron, \qquad j \in s \xrightarrow{+} t \cap BackFron,$$

and

$$f = \operatorname*{argmin}_{x \in ForwFron} SSPE(v, x), \qquad b = \operatorname*{argmin}_{x \in BackFron} SSPE(x, u),$$

where $\operatorname{argmin}_{x \in X} f(x)$ is defined to be the first x in an ordered set X that minimizes $f(x)$. By line 15 in DA, $SSPE(b, u) + length_D(u \to v) + SSPE(v, f) \geq 0$ when DA

stops. Because $SSPE(j, u) \geq SSPE(b, u)$ and $SSPE(v, i) \geq SSPE(v, f)$,

$$\begin{aligned} length_D(j \xrightarrow{+} i) &= SSPE(j, u) + length_D(u \to v) + SSPE(v, i) \\ &\geq SSPE(b, u) + length_D(u \to v) + SSPE(v, f) \geq 0. \end{aligned}$$

By Theorem 6, $D(i) - D(j) \leq length(j \xrightarrow{+} i)$. As $j \xrightarrow{+} i \subseteq s \xrightarrow{+} t$, by Theorem 2, $D(t) - D(s) \leq length(s \xrightarrow{+} t)$, a contradiction. Hence the supposition is false and $\mathcal{K}$ is a cover.

(b) Let $\mathcal{K}_{\min}$ be any minimal cover, and

$$\begin{aligned} I &= \mathcal{K} \cap \mathcal{K}_{\min}, \qquad M = \mathcal{K} - I = \{x | x \in \mathcal{K}, x \notin I\}, \\ E_I &= \{x \to y | x \to y \in E, x \in I, y \in I\}, \\ E_M &= \{x \to y | x \to y \in E, x \in M, y \in M\}, \\ E_{IM} &= \{x \to y | x \to y \in E, x \in I, y \in M, \text{or } x \in M, y \in I\}, \\ E_{\mathcal{K}_{\min}} &= \{x \to y | x \to y \in E, x \in \mathcal{K}_{\min}, y \in \mathcal{K}_{\min}\}. \end{aligned}$$

$\forall x \in M$, when the algorithm is marking $\mathcal{K}$ and x is on top of a frontier priority queue, then the node on top of the other priority queue, denoted as y, must satisfy $y = nil$ (the queue is empty) or $y \in \mathcal{K}_{\min}$ or the x, y pair is in order, because for any path that violates the constraint at least one of its endpoints must be in $\mathcal{K}_{\min}$. So every time the edge value of x decreases, the edge value of some node in $\mathcal{K}_{\min}$ decreases by the same value. In the end x is marked and its edge value reaches zero. Hence $\sum_{x \in M} degree(x) \leq \sum_{y \in \mathcal{K}_{\min}} degree(y)$.

So

$$\begin{aligned} |Touch(M)| &= \sum_{x \in M} degree(x) - |E_M| \\ &\leq \sum_{y \in \mathcal{K}_{\min}} degree(y) - |E_M| \\ &= |Touch(\mathcal{K}_{\min})| + |E_{\mathcal{K}_{\min}}| - |E_M|. \end{aligned}$$

If $M = \emptyset$, $\|\mathcal{K}\| = k$. Otherwise, every node in M is connected directly or indirectly with $\mathcal{K}_{\min}$. $\forall x \in M$, let $x \xrightarrow{+} y$ be the path that prompts the marking of x, then every node in $x \xrightarrow{+} y$ except x and y is already marked, and y must be in $\mathcal{K}_{\min}$. Let $x \to z$ be the first edge in $x \xrightarrow{+} y$.

(a) $z \notin \mathcal{K}_{\min}$, then $z \in M$ because z was marked. So $x \to z$ in E_M.
(b) $z \in \mathcal{K}_{\min}$, then $x \to z$ is a border edge of $\mathcal{K}_{\min}$.

So every node in M is connected by a unique edge in E_M or a border edge of $\mathcal{K}_{\min}$. Hence

$$|M| \leq |E_M| + (|Touch(\mathcal{K}_{\min})| - |E_{\mathcal{K}_{\min}}|).$$

Therefore,

$$\begin{aligned} \|\mathcal{K}\| &= |I| + |M| + |Touch(\mathcal{K})| \\ &= |I| + |M| + |Touch(I)| + |Touch(M)| - |E_{IM}| \end{aligned}$$

$$
\begin{aligned}
&\leq |I| + |E_M| + |Touch(\mathcal{K}_{\min})| - |E_{\mathcal{K}_{\min}}| + |Touch(I)| + \\
&\quad + |Touch(M)| - |E_{IM}| \\
&\leq |I| + |E_M| + |Touch(\mathcal{K}_{\min})| - |E_{\mathcal{K}_{\min}}| + |Touch(I)| + \\
&\quad + |Touch(\mathcal{K}_{\min})| + |E_{\mathcal{K}_{min}}| - |E_M| - |E_{IM}| \\
&= |I| + |Touch(I)| + 2|Touch(\mathcal{K}_{\min})| - |E_{IM}| \\
&\leq k + 2|Touch(\mathcal{K}_{\min})| - |E_{IM}| \\
&< 3k.
\end{aligned}
$$

(c) No node in the cover will be put into a frontier twice, and the algorithm is performing two searches of Dijkstra's algorithm, so the worst-case runtime complexity is $O(k \log k)$. □

Our algorithm can better handle cases that are difficult for the algorithm of Ramalingam et al. Figure 2 shows an example.

If at least one of the frontiers becomes empty, the algorithm can stop as well, since there can be no more conflicts between nodes in different frontiers. As any node is extended if and only if it is in the cover, the cover that DA returns includes all the nonleaf nodes in the two shortest-path trees only. Figure 1 shows the pseudocode for

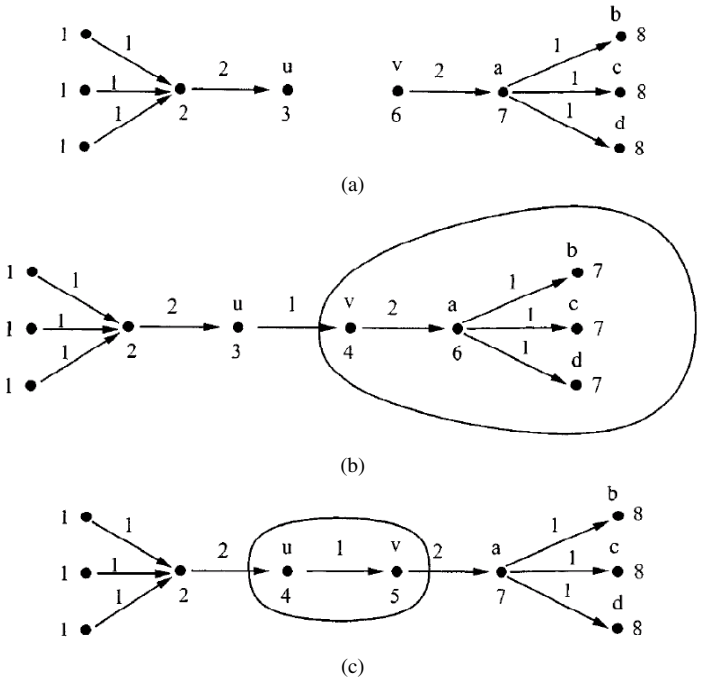


Fig. 2. An example showing an insertion (adapted from [16]). (a) A constraint graph and a feasible solution. (b) The graph after the addition of the constraint $v \rightarrow u \leq 1$, and the updated solution computed by the algorithm of Ramalingam et al. [16]. The values of variables v, a, b, c, and d are modified. (c) The solution computed by our algorithm. Only the values of variables u and v are changed.

DA. According to Dijkstra's algorithm, marked nodes have the following property:

THEOREM 9. *For all nodes x in the constraint graph $G = \langle V, E, length\rangle$, if $x \in$ MarkedForwQ in the algorithm of Figure* 1, *then $SSPE(v, x) = dist_{G_D}(v, x)$; if $x \in$ MarkedBackQ, then $SSPE(x, u) = dist_{G_D}(x, u)$.*

3.2. *The Reassignment Algorithm* (*RA*). After the new edge is inserted in the constraint graph, if it is still feasible, DA will return with two sets that include all the nonleaf nodes in the two shortest-path trees, and two frontier nodes, b and f, which are the nodes with the smallest *SSPE* value in the two frontiers, respectively. If one of the frontiers is empty when DA stops, the corresponding node is set to *nil*. For instance, if the backward frontier is empty, then b is equal to *nil*. This time, we can simply reassign nodes in the backward direction with the shortest-path values and the extended size of actual changes will be smaller than that of the cover because the cover already includes all nodes in the backward direction. In the case where both frontiers are nonempty, b and f provide two bounds for the reassignment, since the path between them represents the tightest constraint between the frontiers.

These two bounds are only two necessary limits, but the reassignment is also restricted by other edges incident with the cover, which complicates things. In order to make better use of the original feasible solution so that the reassignment can be simpler, our reassignment algorithm imposes further restrictions: any node reassigned in the backward direction must be assigned a new value not less than the previous one, any node reassigned in the forward direction must be assigned a new value not greater than the previous one, and for the new constraint between nodes u and v, we enforce $D'(v) - D'(u) = length(u \to v)$.

Figure 3 shows the pseudocode. The reassignment begins from the roots of the two shortest-path trees. Lines 10–17 handle the situation when both frontiers are not empty. In this case $D'(v)$ is set to the larger value of $D(u)+length(u \to v)$ and $D(v)-SSPE(v, f)$.

Figure 4 illustrates this restriction. The space between the two walls represents the region between the forward and backward frontiers, while the boxes represent $u \to v$, and the two shortest-path trees. By Theorem 6, a path violates the constraints if and only if its scaled length is negative, which is represented here as not having enough space. Correspondingly, if the scaled lengths of paths in the graph are positive, which means the values of their endpoints can be squeezed to help allowing in new edges, the situation is represented as having spare space. When $u \to v$ is inserted between the two shortest-path trees with the original node values D, the constraint is violated and in the illustration there is not enough space. If $D(u) + length(u \to v) \geq D(v) - SSPE(v, f)$, then we can reassign values in the forward shortest path tree only, and get the space to satisfy the constraint for $u \to v$ without hitting the bound of $D(v) - SSPE(v, f)$. If $D(v) - SSPE(v, f) > D(u) + length(u \to v)$, then there is not enough space in the forward direction, and we need to reassign nodes in the backward shortest-path tree as well. In either case we keep $D'(v) - D'(u) = length(u \to v)$.

After the solution values for the roots of SP-trees have been updated, every marked node x in the forward SP-tree is reassigned the value $\min(D(x), D'(v) + dist(v, x))$ while every marked node y in the backward SP-tree is reassigned $\max(D(y), D'(u) -$

```
Input:
   G = ⟨V, E, length⟩: the constraint graph;
   D: a feasible solution for G;
   u → v: an edge to insert, which violates D;
   MarkedForwQ and MarkedBackQ: the cover;
   SSPE value: these values satisfy Theorem 9;
   b, f: the frontier outputs when DA stops.
Output:
   D′: the new solution.

 1: Main()
 2:    D′ := D;
 3:    if f = nil then
 4:       D′(v) := D(u) + length(u → v);
 5:       ReassignForw();
 6:    else if b = nil then
 7:       D′(u) := D(v) − length(u → v);
 8:       ReassignBack();
 9:    else
10:       if D(u) + length(u → v) ≥ D(v) − SSPE(v, f) then
11:          D′(v) := D(u) + length(u → v);
12:          ReassignForw();
13:       else do
14:          D′(v) := D(v) − SSPE(v, f);
15:          ReassignForw();
16:          D′(u) := D′(v) − length(u → v);
17:          ReassignBack();
18:
19: ReassignForw()
20:    while MarkedForwQ ≠ empty do
21:       x := MarkedForwQ.pop();
22:       if D′(x) = D(x) then D′(x) := min(D(x), D′(v) + (D(x) + SSPE(v, x) − D(v)));
23:
24: ReassignBack()
25:    while MarkedBackQ ≠ empty do
26:       x := MarkedBackQ.pop();
27:       if D′(x) = D(x) then D′(x) := max(D(x), D′(u) − (D(u) + SSPE(x, u) − D(x)));
```

Fig. 3. The Reassignment Algorithm.

$dist(u, y)$). Although some nodes are in both shortest-path trees, every node value is changed at most once. We have the following theorems:

THEOREM 10. *When RA stops, if $D'(x)$ is set in the forward reassignment, then $D'(x) \leq D(x)$; if $D'(x)$ is set in the backward reassignment, then $D'(x) \geq D(x)$.*

PROOF. (a) x is visited forward.

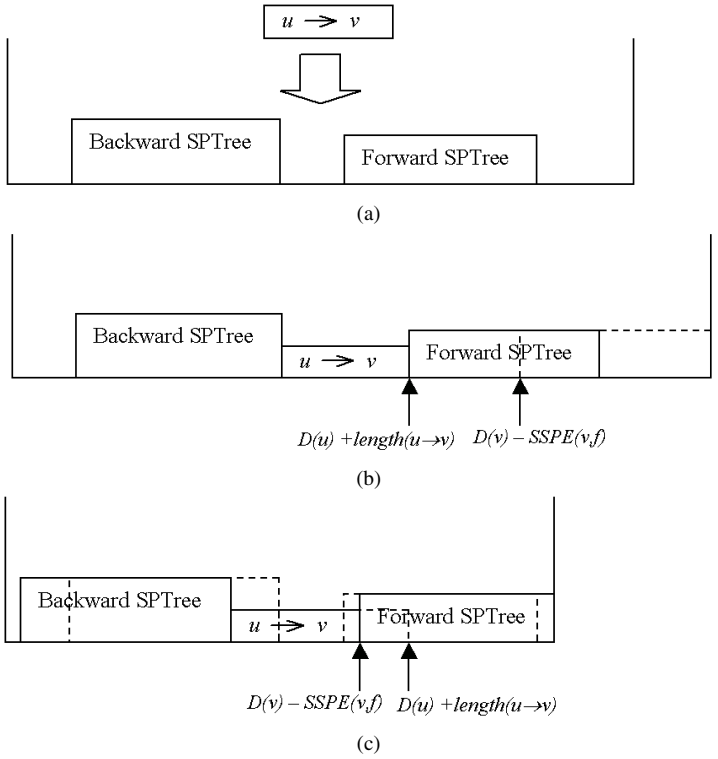


Fig. 4. Illustration of additional restriction. (a) Inserting an edge that violates the constraint. (b) $D(u) + length(u \to v) \geq D(v) - SSPE(v, f)$. (c) $D(v) - SSPE(v, f) > D(u) + length(u \to v)$.

Case 1: $x = v$ *and* $f = nil$. By the algorithm, $D'(v) = D(u) + length(u \to v)$. Because $D(v) - D(u) > length(u \to v)$, $D'(v) = D(u) + length(u \to v) < D(v)$.

Case 2: $x = v$ *and* $f \neq nil$. If $D'(v) = D(u) + length(u \to v)$, then we get the same result as in case 1. If $D'(v) = D(v) - SSPE(v, f)$, then $D'(v) = D(v) - SSPE(v, f) \leq D(v)$ since $v \xrightarrow{*} f$ does not contain any edge that violates the constraint so that

$$\begin{aligned} SSPE(v, f) &= D(v) + length(v \xrightarrow{*} f) - D(f) \\ &= length(v \xrightarrow{*} f) - (D(f) - D(v)) \geq 0. \end{aligned}$$

Case 3: $x \neq v$. By the function ReassignForw (line 19),

$$D'(x) = \min(D(x), D'(v) + (D(x) + SSPE(v, x) - D(v))) \leq D(x).$$

(b) x was visited backward.

Case 1: $x = u$ *and* $f = nil$. $D'(x) = D(u)$.

Case 2: $x = u$ *and* $f \neq nil$. If $b = nil$, then $D'(u) = D(v) - length(u \to v) > D(u)$ since $D(v) - D(u) > length(u \to v)$. Otherwise, by lines 10, 11, and 14 of RA,

$$D'(v) = \max(D(u) + length(u \to v), D(v) - SSPE(v, f)).$$

So $D'(v) \geq D(u) + length(u \to v)$ and by RA (line 16)

$$D'(u) = D'(v) - length(u \to v).$$

Thus

$$\begin{aligned} D'(u) &= D'(v) - length(u \to v) \\ &\geq D(u) + length(u \to v) - length(u \to v) = D(u). \end{aligned}$$

Case 3: $x \neq u$. By the function ReassignBack (line 24),

$$D'(x) = \max(D(x), D'(u) - (D(u) + SSPE(x, u) - D(x))) \geq D(x). \qquad \square$$

If the output from DA is used as input for RA, then RA correctly reassigns the nodes in the cover.

THEOREM 11. *Let $u \to v$ be an edge to be inserted in the constraint graph $G = \langle V, E, length \rangle$, and denote the resulting graph by G'. If G has a feasible solution D, and DA does not detect a negative cycle, then D' computed by our DA and RA is a feasible solution to G'.*

PROOF. For any edge $s \to t \in G'$, if $s \to t = u \to v$, then $D'(v) - D'(u) = length(u \to v)$. If $D'(s) = D(s)$ and $D'(t) = D(t)$, then $D(t) - D(s) \leq length(s \to t)$ since D is a feasible solution to G. Now, we consider the cases where $s \neq u$ or $t \neq v$, and at least one of s and t has been reassigned with a new value.

Case 1: *Only one of s and t is reassigned with a new value.* Let this node be reassigned in the forward direction. If this node is t, then by Theorem 10, $D'(t) - D(s) \leq D(t) - D(s) \leq length(s \to t)$. If this node is s, then $D(t) \leq D'(v) + length(v \xrightarrow{*} t)$, where $v \xrightarrow{*} t$ is the shortest path from v to t through marked nodes. The reason is that if t is in the forward frontier, then because $D'(v) = \max(D(u) + length(u \to v), D(v) - SSPE(v, f))$,

$$\begin{aligned} D'(v) &\geq D(v) - SSPE(v, f) \\ &\geq D(v) - SSPE(v, t) \qquad (f \text{ has the minimal priority}) \\ &= D(v) - (D(v) + length(v \xrightarrow{*} t) - D(t)) \\ &= D(t) - length(v \xrightarrow{*} t). \end{aligned}$$

Otherwise t is in the cover but $D'(t) = D(t)$. By line 22 of RA, $D(t) \leq D'(v) + (D(t) + SSPE(v, t) - D(v)) = D'(v) + length(v \xrightarrow{*} t)$. Thus

$$\begin{aligned} D(t) - D'(s) &= D(t) - (D'(v) + dist(v, s)) \\ &\leq (D'(v) + length(v \xrightarrow{*} t)) - (D'(v) + dist(v, s)) \\ &= length(v \xrightarrow{*} t) - dist(v, s) \\ &\leq length(s \to t). \end{aligned}$$

Case 2: *both nodes are reassigned with a new value*. If both s and t are reassigned in one direction, let this direction be the forward direction. $D'(t) - D'(s) = (D'(v) + dist(v, t)) - (D'(v) + dist(v, s)) \leq length(s \to t)$. If s and t are reassigned in the backward and forward reassignment, respectively, then by Theorem 10, $D'(t) \leq D(t)$ and $D'(s) \geq D(s)$. Thus $D'(t) - D'(s) \leq D(t) - D(s) \leq length(s \to t)$. If s is reassigned in the forward and t is reassigned in the backward reassignment, forming a cycle, then by the fact that no negative cycle exists in G', $dist(v, s) + length(s \to t) + dist(t, u) + length(u \to v) \geq 0$. Thus

$$\begin{aligned} D'(t) - D'(s) &= (D'(u) - dist(t, u)) - (D'(v) + dist(v, s)) \\ &= (D'(u) - D'(v)) - dist(v, s) - dist(t, u) \\ &= -length(u \to v) - dist(v, s) - dist(t, u) \\ &\leq length(s \to t). \end{aligned}$$

□

As a final remark, it is important to note that replacing $D'(v) - D'(u) = length(u \to v)$ with $D'(v) - D'(u) < length(u \to v)$, does not lead to a correct algorithm. The equality is important for checking the cycle constraints, as shown in case 2 of the proof above.

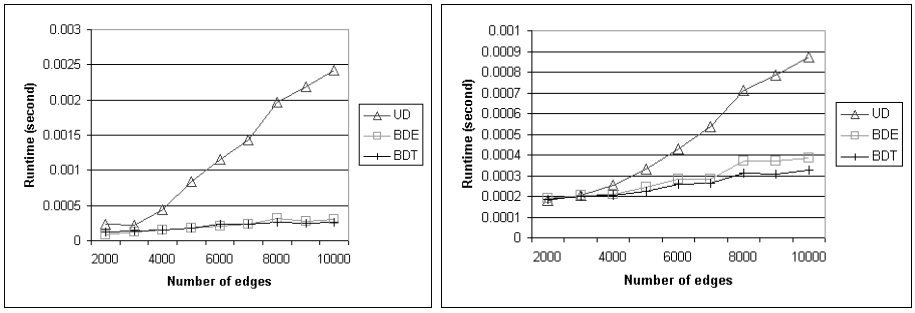
4. Experimental Results. We compare our algorithm and the algorithm of Ramalingam et al. [16] on random graphs. In addition, we implemented a slight variation of our algorithm, which does not use the edge value heuristic but simply extends the forward and backward frontiers in turn. The unidirectional algorithm of Ramalingam et al. is denoted by UD, our bidirectional algorithm using the edge value heuristic by BDE, and its simple turn-based variation by BDT. For uniformity, all the algorithms were implemented under the LEDA [10] framework and use Fibonacci heaps [4] as min-priority queues.

Our test graphs were generated using the random graph generator in the LEDA library. First, we used the potential technique in SPRAND [3] to generate random graphs with negative arc lengths but no negative cycles. We generated an initial feasible solution by an SSoSP algorithm, then one edge of every graph was randomly chosen as the edge to be inserted with a new, smaller length. Let the graph be G, let the feasible solution be D, and let the inserted edge be $u \to v$. The new length of $u \to v$ was set to a random number in the interval

$$\left[\min_{x \in G}(D(x)) - \max_{y \in G}(D(y)) - 1,\ D(v) - D(u) - 1\right].$$

This choice guarantees that the new constraint on $u \to v$ violates the old solution D. All the experiments were performed on a Pentium III 700 MHz workstation with 512 MB of memory.

Figure 5 shows the performance of the three algorithms. At each data point, 10,000 feasible graphs were randomly generated. The number of nodes was set to 1000 and the number of edges varies from 2000 to 10,000. On each graph, a batch of 20 experiments with randomly created single edge insertions was run. After an insertion, some graphs



(a) (b)

Fig. 5. Average runtime for a batch of 20 insertions. G' with (a) and without (b) negative cycles.

contained negative cycles while others were still feasible. For graphs with negative cycles, the algorithm detects the cycle and stops. For feasible graphs, the algorithm computes a new feasible solution. We measure the performance of these two cases separately. The number of graphs that contain negative cycles after an insertion is shown in Table 1. The percentage of such graphs increases with the number of edges.

We also compare the runtime of incremental algorithms with that of a nonincremental approach using the Bellman–Ford algorithm on augmented constraint graphs. Figure 6 shows the results. The speed-up factor is defined as

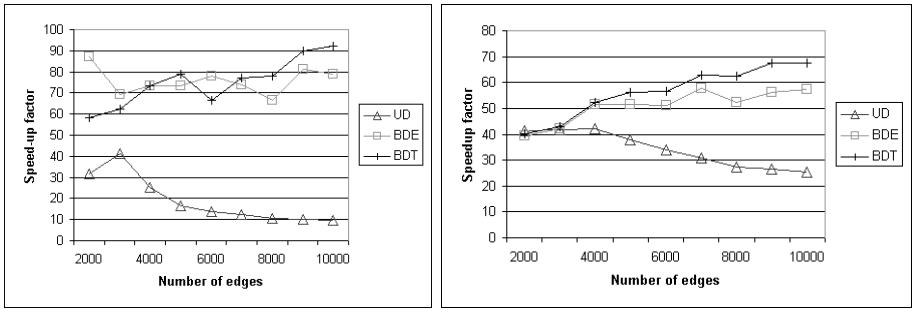
$$\frac{\text{runtime(nonincremental algorithm)}}{\text{runtime(incremental algorithm)}}.$$

Similar to Figure 5, speed-up factors on G' with and without negative cycles are measured separately. While both parts indicate that the speed-up factor of UD has a trend of declining in dense graphs, Figure 6(a) shows that both versions of bidirectional search can maintain their speed-up factors if there are negative cycles and Figure 6(b) even indicates an improvement in the remaining majority cases without negative cycles. Between BDE and BDT, BDT performs better.

Figure 7 shows the cover size $|\mathcal{K}|$ and extended size $\|\mathcal{K}\|$ of the computed covers. UD reassigns all nodes in the cover that it marks. For BDE and BDT, our reassignment algorithm, the number of actual changes can be smaller than the cover size, as shown in the discussion of RA. In all experiments, UD marked much larger covers than BDE and BDT. The difference of cover sizes between BDE and BDT is slight in our tests. In an

Table 1. Among the 200,000 graphs resulting from single edge insertions, numbers of cases that contain negative cycles

Number of edges	2000	3000	4000	5000	6000	7000	8000	9000	10,000
Number of graphs with NC	473	1,386	3,752	8,401	15,326	23,727	32,481	41,772	50,963
Percentage in 200,000 graphs	0.2	0.7	1.9	4.2	7.7	11.9	16.2	20.9	25.5



(a) (b)

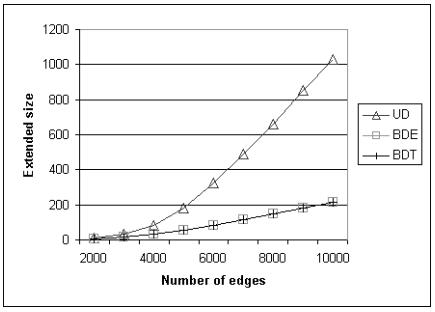
Fig. 6. Speed-up factors for a batch of 20 insertions. G' with (a) and without (b) negative cycles.

application, if changes of node values are associated with some expensive operations, the number of actual changes necessary is of vital importance.

Figure 8 plots the performance when the batch size varies. In Figure 8(a), at each data point, we generate 10,000 random graphs with 5000 edges while the batch size varies from 1 to 19. Each incremental change in the batch adds a single edge to the *same* original graph G. The result illustrates the tradeoff between one-time costs, such as initializing data structures in LEDA, and performing the actual incremental algorithm. As the size of the batch increases, the performance of BDE and BDT increases faster than that of UD, due to larger initial overheads in BDE and BDT. In Figure 8(b), in order to show the performance over a sequence of updates to a given system, we change the

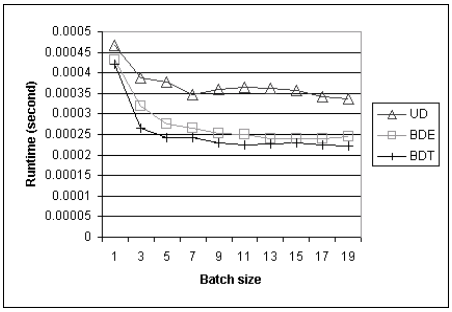
Number of edges	2000	3000	4000	5000	6000	7000	8000	9000	10000
$\|\mathcal{K}\|$ of UD	3.189	6.962	14.931	26.490	39.984	52.608	63.188	72.984	80.214
$\|\mathcal{K}\|$ of BDE	2.330	3.601	5.354	7.330	9.263	10.982	12.306	13.510	14.443
Act. Chg. of BDE	1.915	3.067	4.690	6.549	8.379	10.002	11.249	12.380	13.257
$\|\mathcal{K}\|$ of BDT	2.349	3.626	5.371	7.340	9.267	10.985	12.301	13.502	14.431
Act. Chg. of BDT	1.963	3.125	4.747	6.597	8.424	10.045	11.280	12.412	13.285

(a) Cover size $|\mathcal{K}|$

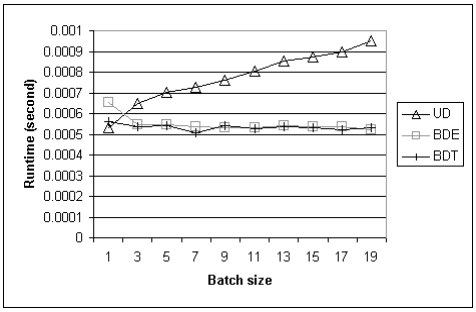


(b) Extended cover size $\|\mathcal{K}\|$

Fig. 7. The size (a) and extended size (b) of the cover.



(a)



(b)

Fig. 8. Performance with varied batch sizes. (a) Independent single insertions applied to the same given graph, illustrating the tradeoff between one-time costs and performing the actual incremental algorithm. (b) A sequence of several insertions in a row applied to a graph, showing how a difference constraint system evolves under updates.

definition of *batch* a little while keeping the other settings of the experiment. Here, a *batch* is a sequence of edges to be inserted to a given random graph, and every time a new edge is inserted, we apply the incremental algorithms to update the solution. If negative cycles are detected, we delete the newly inserted edge and perform the next insertion; otherwise we go directly to the next insertion. Compared with Figure 8(a), the result of (b) indicates that when insertions are applied consecutively to a system of difference constraints, incremental updates seem to become more difficult, as we can see that the declining trend of the runtime in Figure 8(a) has gone. For UD, the situation is especially bad, as is reflected by the increasing average runtime in Figure 8(b). BDE and BDT are doing much better. Their average runtime stays constant, or even improves slightly as the batch size gets bigger. This indicates that the performance gain from amortizing the one-time costs seems to be about the same as the performance loss caused by an increase in difficulty of solving the evolving systems with BDE and BDT.

Figure 9 shows the speed-up over the nonincremental Bellman–Ford algorithm with the new definition of batches (the speed-up being less than that of Figure 6 is due to the effect mentioned above that incremental algorithms are making the system harder to update). Again, this result confirms the outstanding performance of BDE and BDT in our tests.

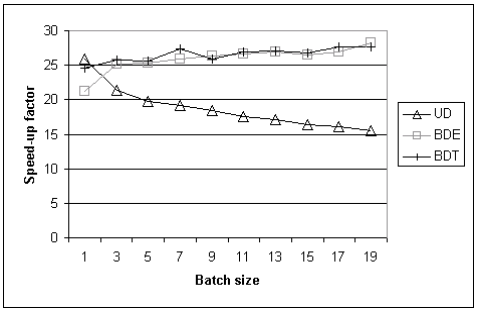


Fig. 9. Speed-up factor with the new definition of batches.

Overall, BDE and BDT outperform UD by a large margin. Further, the results show that although the edge value heuristic is desirable in theory, since it leads to a bounded extended size of the cover and bounded runtime complexity in the worst case, in practice the simple strategy of alternating the search in the two directions can compute covers of almost the same size, and can execute faster due to smaller overhead. Our results show that bidirectional search can outperform unidirectional search for incremental difference constraint problems.

5. Extensions. Bidirectional search has regained the attention of researchers in recent years [8], [9]. However, much of the work has focused on single-agent search on large graphs, and deals with the problem of reducing the memory requirements of such algorithms. Our result shows that another interesting application of bidirectional search algorithms lies in relatively smaller search spaces such as incremental updates of graphs, where the average overheads decrease as batch sizes increase and other criteria such as cover size are important.

As discussed in [16] and Section 2, the requirement of SDC is weaker than that of SSoSP. Hence incrementally maintaining an SDC may serve as a better underlying system than maintaining SSoSP solutions in some graph update problems, such as incremental negative cycle detection. Our results indicate that for such kinds of problems, an approach using incremental SDC with bidirectional search can outperform incremental SSoSP with its unidirectional forward search.

Other possible extensions include more elaborate versions of the algorithm. In our algorithm we allow the overlapping of the frontier searches in the forward and backward direction which can lead to marking some nodes twice. It is an interesting question whether this can actually happen. More importantly, it is an open question how to handle the interactions between all the searches when multiple insertions are introduced simultaneously. In addition, considering the size of the forward and backward SP-trees in RA may further reduce the number of actual changes.

Acknowledgements. We thank an anonymous referee for valuable suggestions that significantly improved the presentation of this paper.

References

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

[2] A. Berman, M. Paull, and B. Ryder, Proving relative lower bounds for incremental algorithms, *Acta Informatica*, **27** (1990), 665–683.

[3] B. Cherkassky and A. Goldberg, Negative-cycle detection algorithms, *Proceedings of the Fourth Annual European Symposium on Algorithms*, pp. 349–363, 1996.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (second edition), MIT Press, Cambridge, MA, 2001.

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

[6] D. Dreyfus, An appraisal of some shortest path algorithms, *Operations Research*, **17** (1969), 395–412.

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

[8] H. Kaindl and G. Kainz, Bidirectional heuristic search reconsidered, *Journal of Artificial Intelligence Research*, **7** (1997), 283–317.

[9] R. Korf, Divide-and-conquer bidirectional search: first results, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (*IJCAI*-99), pp. 1184–1189, 1999.

[10] K. Mehlhorn and S. Näher, LEDA: a platform for combinatorial and geometric computing, *Communications of the ACM*, **38** (1995), 96–102.

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

[12] I. Pohl, Bi-directional search, *Machine Intelligence*, **6** (1971), 127–140.

[13] V.R. Pratt, Two easy theories whose combination is hard, Technical report, Massachusetts Institute of Technology, September 1977. Also available online: http://boole.stanford.edu/pub/sefnp.pdf.

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

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

[16] G. Ramalingam, J. Song, L. Joskowicz, and R. E. Miller, Solving systems of difference constraints incrementally, *Algorithmica*, **23** (1999), 261–275.

[17] S. Russell and P. Norvig, *Artificial Intelligence*: *A Modern Approach*, Prentice-Hall, Upper Saddle River, NJ, 1995.