On the Distributed Bellman-Ford Algorithm and the Looping Problem

Kevin R. Hutson, Terri L. Schlosser, Douglas R. Shier,

INFORMS Journal on Computing
Vol. 19, No. 4, Fall 2007, pp. 542–551
ISSN 1091-9856 | EISSN 1526-5528 | 07 | 1904 | 0542
DOI 10.1287/ijoc.1060.0195
© 2007 INFORMS

On the Distributed Bellman-Ford Algorithm and the Looping Problem

Kevin R. Hutson
Department of Mathematics, Furman University, Greenville, South Carolina 29613, kevin.hutson@furman.edu

Terri L. Schlosser
IBM Tivoli Software, Austin, Texas 78758, tschloss@us.ibm.com

Douglas R. Shier
Department of Mathematical Sciences, Clemson University, Clemson, South Carolina 29634, shierd@clemson.edu

The classic Bellman-Ford algorithm for calculating shortest paths can be easily adapted to a distributed environment in which the computations are performed locally by identical processors at each network node. A distributed shortest-path algorithm is particularly appropriate for use in communication networks to capitalize on local information rather than rely on a central controller. This paper discusses the behavior of a synchronous version of the distributed Bellman-Ford algorithm in a dynamic environment in which communication link costs can undergo change. Several algorithms are described that mitigate or eliminate the occurrence of *looping*, which is responsible for degrading the performance of distributed shortest-path algorithms. We provide theoretical and computational evidence to show that two proposed algorithms offer improvements upon the original and modified Bellman-Ford algorithms.

Key words: adaptive routing; communication networks; distributed computing; looping; shortest paths; trees
History: Accepted by Prakash Mirchandani, Area Editor for Telecommunications and Electronic Commerce; received August 2005; revised May 2006; accepted June 2006. Published online in *Articles in Advance* July 20, 2007.

1. Introduction

Shortest-path problems arise in a variety of situations in which goods, messages, or vehicles need to be efficiently routed between locations (Ahuja et al. 1993). In particular, shortest-path algorithms have been incorporated into the protocols that direct the routing of calls and data through modern communication networks. In the context of communication networks, *distributed* shortest-path algorithms are particularly important because they allow each node to make routing decisions based on local (rather than global) topological information. Reliance on a central controller, which is subject to failure, is thus avoided. Moreover, changes in topology or cost data can be identified locally and then transmitted to the rest of the network, together with updated routing information. Efficient distributed shortest-path algorithms that can gracefully accommodate such changes are desired.

A classic shortest-path algorithm is the Bellman-Ford algorithm (Bellman 1958). A distributed version of this algorithm is easily constructed and forms the basis of the original routing algorithm used in the ARPANET as well as in the MERIT, TIDAS, and NSFNET network architectures (Cegrell 1975, Garcia-Luna-Aceves 1988, Johnson 1984, Schwartz 1986). This distributed algorithm has been shown to converge in a finite number of steps if all cycles in the network have positive cost and the arc costs are fixed (Bertsekas and Gallager 1987, Bertsekas and Tsitsiklis 1989). However, if some arc costs increase, a looping behavior can occur, in which messages revisit the same node on the way to their destination. Such looping causes convergence to be very slow (Humblet 1991, Jaffe and Moss 1982, Schwartz 1986).

In this paper several variations of the distributed Bellman-Ford algorithm are discussed and analyzed. These variations have been developed to address the problem of looping when arc costs are increased (e.g., when network resources go down). There is considerable literature on approaches to mitigate and eliminate looping in the context of store-and-forward protocols for transmitting data in computer communication networks. (Garcia-Luna-Aceves 1988, 1993, 1997; Humblet 1991; Jaffe and Moss 1982; Merlin and Segall 1979; and Shin and Chou 1993). Typically each node i maintains a *neighbor table* (*distance table*) and a *routing table*: The former stores shortest-path information via each node j adjacent from the given node i, while the latter stores such information for node i itself. The objective is to send each message to the next node along the shortest path to its destination. To avoid revisiting a node as a message is forwarded from node to node, a more complex protocol needs to be enforced (Garcia-Luna-Aceves 1997, Humblet 1991, Jaffe and Moss 1982, Merlin and Segall 1979, Shin and Chou 1993).

Our solution differs from the above approaches in several ways. First, we focus on the single-origin shortest-path problem (rather than considering different destinations) and therefore do not need the additional storage required for maintaining neighbor tables. Second, we study the synchronous version of the distributed shortest-path problem which, as will be seen (Section 3), introduces some subtle difficulties by virtue of the simultaneous selection of routes. Also our approach applies more generally to directed (rather than undirected) networks and can accommodate negative arc costs, so long as all cycles have positive cost. Finally, our approach is based on the natural idea of an "ancestor list" and this enables a fairly crisp and simple analysis of algorithm correctness, compared to more laborious proofs in the literature. The emphasis on modifying the Bellman-Ford algorithm for a distributed environment also distinguishes our approach from related work in the computer-science literature (Cicerone et al. 2003, Ramalingam and Reps 1996, Ramarao and Venkatesan 1992).

Section 2 presents background on the Bellman-Ford algorithm and its standard distributed implementation. The problem of looping, and some previous attempts to mitigate its effects, are discussed in Section 3. In addition, we introduce the ancestor-list algorithm and provide a compact analysis of its convergence properties. This algorithm is extended to a new type of loop-free algorithm in Section 4, where we easily establish its validity and complexity. Computational results given in Section 5 indicate that our proposed ancestor-list and loop-free algorithms can improve on the original distributed Bellman-Ford algorithm, sometimes in a dramatic fashion.

2. Background

We consider a directed network $G = (N, A)$ with node set N and arc set A. Each arc $(j, k) \in A$ has an associated cost (length) c_{jk}. Let $A(j)$ denote the set of nodes adjacent from node j and let $B(j)$ denote the set of nodes adjacent to node j. A *path* P is a sequence of arcs $(i_0, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k)$ in G; it can also be denoted by the node sequence $[i_0, i_1, \ldots, i_k]$. Such a path P has origin node i_0 and destination node i_k. A *cycle* is a path with equal origin and destination nodes. The cost $c(P)$ of path P is the sum of all c_{jk} with $(j, k) \in P$. A *shortest path* from node s to node t is a path of minimum cost from s to t.

We concentrate here on the *single-source shortest-path problem*—finding shortest paths from the source node s to every other node of network G. Indeed, the single-source problem forms the basis for virtually all point-to-point routing algorithms used in communication networks (Schwartz 1986). We assume that all nodes are reachable by a directed path from s. To ensure that shortest paths exist in G, we also assume that any cycles in G have positive cost. In this case, we define the shortest-path distance $d(j)$ to be the cost of a shortest path from s to j. These shortest-path distances must satisfy the following optimality conditions, known as Bellman's equations:

$$\begin{aligned} &d(s) = 0 \\ &d(j) = \min\{d(i) + c_{ij} \colon i \in B(j)\} \quad \text{for all } j \neq s. \end{aligned} \tag{1}$$

These conditions state that the cost of a shortest path from s to j is equal to the minimum of the shortest-path distance from s to i plus the cost of the arc (i, j) where i is adjacent to j.

One algorithm that attempts to satisfy equation (1), using an iterative approach, is the Bellman-Ford algorithm (Bellman 1958). It maintains at each iteration k a vector D^k of size n, where $n = |N|$ is the number of nodes in network G; $D^k(j)$ is the current distance label for node j. At the beginning of the algorithm, $D^0(s)$ is initialized to 0 and $D^0(j) = \infty$ for all $j \neq s$. At each iteration the vector D^k is updated using (1) until the distance vectors converge to $D^* = (d(1), \ldots, d(n))$ and Bellman's equations have been satisfied. The shortest-path tree rooted at node s is maintained using a predecessor array *pred*. Pseudocode for the Bellman-Ford algorithm is given below.

```
bellman-ford
    $D^0(s) := 0$; pred$(s) := 0$; $h := 0$;
    $D^0(j) := \infty$ for all $j \neq s$;
    do
        for all $j \neq s$
            $D^{h+1}(j) := \min\{D^h(i) + c_{ij} \colon i \in B(j)\}$;
            pred$(j) := i$ such that $D^{h+1}(j) = D^h(i) + c_{ij}$;
        end for;
        $D^{h+1}(s) := 0$; $h := h + 1$;
    while $D^h \neq D^{h-1}$
```

An inductive argument can be used to show that D^k represents shortest-path distances using paths having at most k arcs (Bertsekas and Gallager 1987). Because no shortest path can contain more than $n-1$ arcs, the Bellman-Ford algorithm converges in at most $n-1$ iterations, with the iterates converging monotonically: $D^h(j) \geq D^{h+1}(j)$ for all j. Because each arc (i, j) is processed at most once per iteration, the Bellman-Ford algorithm has worst-case complexity $O(mn)$, where $m = |A|$. An inductive argument can also be used to show convergence of the algorithm, in at most n iterations, if D^0 is initialized to values at least as large (componentwise) as the shortest-path distances D^*; however in this case convergence to D^* need not be monotonic.

In the standard Bellman-Ford algorithm, the current information D^k is maintained and updated by a centralized controller, requiring a central database that knows the cost structure and node information for the entire network. In communication networks, however, it is difficult to implement such a centralized device. Therefore, a decentralized version of the distributed Bellman-Ford algorithm stores information at the nodes, rather than at some centralized location. Each node j maintains the cost of all arcs incident to j, the identity of all nodes adjacent from j, the current predecessor pred(j), and the current distance $D(j)$ from the source node s. This allows each node j to receive the current distance labels from nodes adjacent to j, update its own distance label and predecessor node, and then send its current distance to all nodes adjacent from j. Pseudocode for a synchronous version of the distributed Bellman-Ford algorithm is given below. For added clarity, we have included the variable *count* that tracks the number of nodes that have updated their distance labels during each iteration of the algorithm. The algorithm terminates when $D'(j) = D(j)$ for all j. Convergence is again guaranteed in at most $n-1$ iterations, and so the distributed Bellman-Ford algorithm has worst-case complexity $O(mn)$.

```
distributed bellman-ford
  $D(s) := 0$; pred$(s) := 0$; count $:= 1$;
  $D(j) := \infty$ for all $j \neq s$;
  while (count $> 0$)
    count $:= 0$;
    for all $j$   //send information
      send $D(j)$ to all nodes $k \in A(j)$;
    end for;
    for all $j \neq s$   //compute labels
      $D'(j) := \min\{D(i) + c_{ij}: i \in B(j)\}$;
      if $D'(j) < D(j)$ then
        count $:=$ count + 1;
        $D(j) := D'(j)$;
        pred$(j) := i$ such that $D'(j) = D(i) + c_{ij}$;
      end if;
    end for;
  end while
```

Figure 1 shows an application of the distributed Bellman-Ford algorithm to find shortest paths in an undirected network with source node $s = 1$. (Note that any undirected network can be represented as a directed network with arcs (i, j) and (j, i) having identical cost.) At each iteration the directed tree defined by the predecessor array is shown as well as the current label $D(j)$ for each node j.

The original undirected network is given in Figure 1(a), together with the initial values of $D(j)$. At the first iteration, shown in Figure 1(b), nodes 2 and 4 update their information to $D(2) = 1$, pred$(2) = 1$, $D(4) = 4$, and pred$(4) = 1$; the information at node 3 remains unchanged. At the next iteration, node 3 performs the updates $D(3) = 2$ and pred$(3) = 2$; see

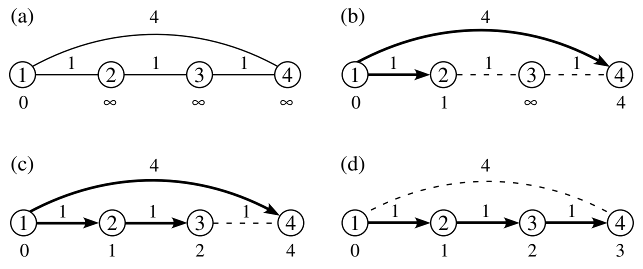

Figure 1 Example of Distributed Bellman-Ford

Figure 1(c). Then in Figure 1(d), node 4 updates its information, producing $D(4) = 3$ and pred$(4) = 3$. At the next step the information at all nodes remains unchanged, so the algorithm terminates.

3. Ancestor-List Algorithm

The previous description of the distributed Bellman-Ford algorithm assumes that the arc costs remain stable. However, the regular changes occurring in communication networks cast doubt on this assumption. Suppose that after shortest paths in a network have been found some arc cost c_{ij} changes. If we then restart the algorithm using the previous optimal node labels, the speed of convergence of the algorithm will be highly dependent on the new c_{ij}. If the arc cost decreases from its previous value then convergence of the algorithm is still guaranteed within n iterations, since the previous optimal distance labels are at least as large as the new shortest-path distances. If the arc cost increases then finite convergence of the algorithm is still guaranteed, assuming all cycles have positive cost, but the number of iterations need no longer be polynomially bounded in the network size (Bertsekas and Tsitsiklis 1989). This is due to a "looping" behavior (Garcia-Luna-Aceves 1988, Jaffe and Moss 1982, Shin and Chou 1993) of the updates in the network.

An illustration of such looping is shown using the same (undirected) network as in Figure 1, repeated in Figure 2(a). The previously found shortest-path tree, rooted at $s = 1$, is displayed in Figure 2(b), together with the optimal node labels $d(x)$. Now suppose that arc $(1, 2)$ goes down, or equivalently, $c_{12} = \infty$. Then node 2 carries out an update using the current information, producing $D(2) = 3$ and pred$(2) = 3$ in Figure 2(c). Notice that the *predecessor graph* defined by the arcs $\{(\text{pred}(x), x)\}$ is no longer a tree: it contains the cycle $[2, 3, 2]$. Updates to node labels continue until Figure 2(f) where the precedessor graph now again becomes a tree, although not all of the distance labels are correct path distances. After one more step, node 2 is updated using node 3 and the distance labels now represent the correct shortest-path distances $D^* = (0, 6, 5, 4)$. Notice however that for several iterations, some nodes y updated their labels $D(y)$ using nodes x that had y as their predecessor,

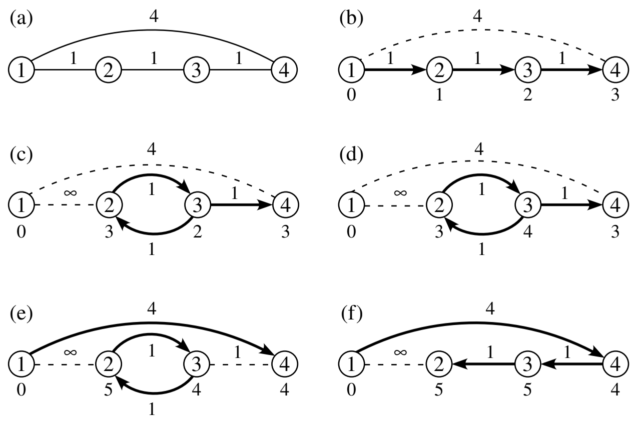

Figure 2 **Example of Looping Behavior**

and this looping behavior impeded the progress of the algorithm.

An alternative algorithm proposed to reduce such looping behavior is the *predecessor algorithm* (Cegrell 1975, Schwartz 1986, Shin and Chen 1987). This algorithm adds one condition to the update step of the distributed Bellman-Ford algorithm. Namely, if $\text{pred}(x) = y$ then node y should ignore the contribution $D(x) + c_{xy}$ in performing its update. Because node x maintains the information $\text{pred}(x)$, it becomes the responsibility of node x to refrain from sending the (incorrect) label $D(x)$ to its parent node y; for example, it can send $D(x) = \infty$ instead. When the predecessor algorithm is applied to the network in Figure 2, looping behavior is completely avoided.

The predecessor algorithm allows node y to avoid looping caused by its children in the current tree. However, looping can still occur with respect to other nodes x such that node y is an ancestor of node x in the current tree (Garcia-Luna-Aceves 1988, Shin and Chen 1987). Therefore we propose a more inclusive approach, called the *ancestor-list algorithm*, to improve further the performance of the distributed Bellman-Ford algorithm. We assume that the shortest-path distances $d(x)$ from node s have been previously calculated and that the arc cost c_{ij} increases for some arc (i, j), triggering an increase in $D(j)$.

The ancestor-list algorithm requires each node x to maintain a list $AL(x)$ of nodes. Once the algorithm has converged to the new shortest-path distances, $AL(x)$ will represent the set of ancestors of node x in the new shortest-path tree. The ancestor-list algorithm (whose pseudocode is given below) requires passing some additional information, namely $AL(x)$, from node x to all nodes $y \in A(x)$. Also it adds one condition to the update step of the distributed Bellman-Ford algorithm. Namely, if node $y \in AL(x)$, then an update using arc (x, y) is prohibited; equivalently, this can be achieved by node x sending the label $D(x) = \infty$ to node y. In this way node y avoids looping relative to any descendant node x in the current tree.

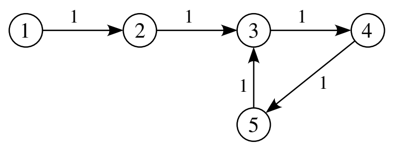

Figure 3 **Looping Example for the Predecessor Algorithm** ($s = 1$)

For example, when the cost of arc $(1, 2)$ is increased to $c_{12} = M \geq 5$ in Figure 3, the ancestor-list algorithm produces no looping and requires 5 iterations to terminate. Yet the original Bellman-Ford and predecessor algorithms incur looping and require $O(M)$ iterations.

```
ancestor list
  $D(x) := d(x)$ for all $x$; count := 1;
  create $AL(x)$ for all $x$ using the current
    predecessor graph;
  **while** (count > 0)
    count := 0;
    **for** all $x$   //send information
      send $D(x)$, $AL(x)$ to all nodes $y \in A(x)$;
    **end for**;
    **for** all $y \neq s$   //compute labels
      $D'(y) := \min\{D(x) + c_{xy}\colon x \in B(y),\ y \notin AL(x)\}$
             $= D(r) + c_{ry}$;
      $D(y) := D'(y)$; $\text{pred}(y) := r$;
        $AL(y) := AL(r) \cup \{r\}$;
      **if** $D'(y) \neq D(y)$ or $r \neq \text{pred}(y)$ **then**
        count := count + 1;
    **end for**;
  **end while**
```

The ancestor-list algorithm does not completely prevent looping. The difficulty is that computations are taking place simultaneously at each node. Therefore it is possible for some node y to update $D(y)$ and $\text{pred}(y)$ using a node x that is not a descendant of y in the current tree T; however another node w may be *simultaneously* updating its own label $D(w)$ and $\text{pred}(w)$ from node v. As seen in Figure 4, this can in certain circumstances produce a cycle $[x, y, \ldots, v, w, \ldots, x]$ in the new predecessor graph.

Although the ancestor-list algorithm does not completely prevent looping, we can show that the amount of looping is polynomially bounded. This is in contrast to the predecessor algorithm (as well as the original distributed Bellman-Ford algorithm) whose worst-case complexity is pseudopolynomial, involving the magnitude of the increased arc cost c_{ij}. To establish an $O(n)$ bound on the number of iterations required by the ancestor-list algorithm, we present some additional notation and preliminary lemmas.

Suppose that at step $k = 0$ the increased cost c_{ij} is detected at node j. At any step $k \geq 0$, let $AL_k(y)$ denote the current ancestor list of node y. We then have the following result.

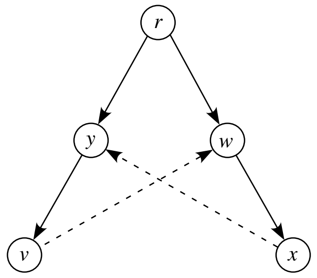

Figure 4 **Creation of a Cycle Using the Ancestor-List Algorithm**

Lemma 1. *If the update* (x, y) *occurs at step* k, *then* $|AL_{k+1}(y)| = |AL_k(x)| + 1$.

Proof. Because node y was updated from node x at step k, we have $AL_{k+1}(y) = AL_k(x) \cup \{x\}$. We claim then that $x \notin AL_k(x)$. Suppose to the contrary that $x \in AL_k(x)$. Then $AL_k(x) = AL_{k-1}(v) \cup \{v\}$ for some node $v \neq x$, so we would have $x \in AL_{k-1}(v)$. However, this means that x is in the ancestor list of v at step $k-1$ so the update (v, x) is prohibited by the ancestor-list algorithm. This contradiction shows that $x \notin AL_k(x)$ and the stated conclusion follows. □

Without loss of generality we can assume that the increase in cost c_{ij} occurs for an arc $(i, j) \in T$, the original shortest-path tree. Thus the only possible changes to the shortest-path tree T will occur within the subtree T_j of T, rooted at node j. Consequently, at any step k let us partition T_j into the three sets F_k, I_k, and R_k. The nodes in F_k are nodes of T_j having a *feasible* label at step k. This concept is defined inductively: at step $k = 0$ all nodes of $T - T_j$ have feasible labels as well as node j; a node y has a feasible label at step k if it was updated from a node x having a feasible label at step $k-1$. In essence, feasible labels represent valid path lengths from node s in the current network (with increased c_{ij}) and thus are upper bounds on the true shortest-path distances. The nodes in I_k are those with infinite labels, while R_k denotes the remaining nodes of T_j. (The appendix shows that the set I_k can, in certain circumstances, be nonempty.) Clearly, nodes of I_k cannot update any other nodes at step k. Explicitly we assume that at step $k = 0$, node j detects the increase in cost c_{ij} and updates its distance label $D(j)$; thus $F_0 = \{j\}$ and $R_0 = T_j - \{j\}$.

Lemma 2. $|AL_k(y)| \geq k$ *for all* $y \in R_k$.

Proof. The result clearly holds for $k = 0$ since every node of $T_j - \{j\} = R_0$ has a predecessor in T_j. Suppose it holds at step k: i.e., $|AL_k(v)| \geq k$ for all $v \in R_k$. Now select $y \in R_{k+1}$; it was updated from some node x at step k. Notice that if $x \in F_k$ or $x \in T - T_j$ then we would have $y \in F_{k+1}$. Consequently, we have $x \in R_k$ and by induction $|AL_k(x)| \geq k$. Then by Lemma 1, $|AL_{k+1}(y)| = |AL_k(x)| + 1 \geq k + 1$, completing the inductive proof. □

We can now show that, while the ancestor-list algorithm may cycle, it is nonetheless guaranteed to converge in $O(n)$ iterations, resulting in $O(mn)$ worst-case complexity. Interestingly enough, this is the same order of complexity as the original Bellman-Ford algorithm, even though the ancestor-list algorithm now works in a distributed and dynamic environment.

Theorem 1. *The ancestor-list algorithm converges in at most* $3n$ *iterations to the correct shortest-path distances after an increase occurs to* c_{ij}, *resulting in* $O(mn)$ *worst-case complexity.*

Proof. By Lemma 2, at step n any node $v \in R_n$ would have $|AL_n(v)| \geq n$, which is clearly impossible. However there may exist nodes $v \in I_n$, though any such node v cannot update any other nodes at the next iteration. Since T_j is strongly connected from node j, there is some node $y \in I_n$ adjacent from a node $x \in F_n$. Then at the next iteration $y \in F_{n+1}$. Moreover, $x \in F_{n+1}$ since node x cannot be updated from any node in I_n. (We note that it is possible at some earlier step $k < n$ for $x \in F_k$ to move to $x \in R_{k+1}$ after being updated by some node $y \in R_k$.) Thus F_{n+1} strictly contains F_n. Continuing in this way, we obtain $F_n \subset F_{n+1} \subset F_{n+2} \subset \cdots$ so that $T_j = F_r$ must hold for some $r \leq 2n$. At this point all nodes in the network have feasible labels, representing upper bounds on the true shortest-path distance from s, and so convergence is assured in at most n more iterations. Altogether, at most $3n$ iterations are required by the ancestor-list algorithm. Using an appropriate data structure, the ancestor lists $AL(y)$ can be updated in constant time, so there is $O(m)$ work per iteration, giving $O(mn)$ worst-case complexity. □

We remark that the results obtained here apply more generally to simultaneous cost changes to multiple arcs $(i_1, j_1), (i_2, j_2), \ldots, (i_p, j_p)$. Namely, we can use T instead of T_j as our base tree and again partition T into the sets F_k, I_k, R_k. Initially F_0 contains any node not in $T_{j_1} \cup T_{j_2} \cup \cdots \cup T_{j_p}$ and $R_0 = T - F_0$. Then the proofs of Lemma 1, Lemma 2, and Theorem 1 remain valid, giving us polynomial time complexity for the ancestor-list algorithm.

4. A Loop-Free Algorithm

At the expense of extra data transfer and only a slightly more involved protocol to be followed at each node, the ancestor-list algorithm can be modified to avoid looping altogether. A variety of strategies have been proposed to maintain a "loop-free" routing; see Garcia-Luna-Aceves (1988, 1993, 1997), Humblet (1991), Jaffe and Moss (1982), Merlin and Segall (1979), and Shin and Chou (1993). In the approach taken here, each node x keeps and updates an ancestor list $AL(x)$. Suppose now that an increase in cost occurs on an arc incident to a particular node j. If this does not trigger an increase in the label $D(j)$, the algorithm proceeds as in the standard distributed Bellman-Ford algorithm. Otherwise, a different protocol is now initiated at node j. Namely, node j sends, in addition to $D(j)$ and $AL(j)$, the special message $[j, token]$ to all nodes $y \in A(j)$, signifying the identity of the root j of the subtree T_j of T directed from j. The function of the token will be discussed later. Let $G_j = (N_j, A_j)$ denote the subgraph of G induced by the nodes N_j of T_j.

Similar to Garcia-Luna-Aceves (1988), Jaffe and Moss (1982), and Merlin and Segall (1979), our objective is to limit the update of information within the subgraph G_j to avoid the creation of cycles, as occurs in Figure 4. To accomplish this, we do not allow the inaccurate information at nodes $w \in N_j$ to be used once the updated node label information (sent down through G_j from its root j) has reached node w. First, node j declares itself "trustworthy." It then passes a token to all $y \in A(j)$ so that at the next time step those nodes $y \in N_j$ can declare themselves trustworthy and continue forwarding the special message $[j, token]$ as well as $D(y)$ and $AL(y)$. Notice that, if a token is received by some $y \in N$, then node y can determine whether it is in N_j by checking if $j \in AL(y)$. Thus the special message can be seen as spreading in a breadth-first fashion from j throughout the subgraph G_j. Our idea refines the approaches in Jaffe and Moss (1982) and Merlin and Segall (1979), which attempt to "freeze" nodes after an update has been detected, thus not allowing old information to be used until the node is "unfrozen." In Garcia-Luna-Aceves (1988, 1993, 1997), various mechanisms are used to screen the neighboring nodes to determine the reliability of the information being passed.

Once a node y becomes trustworthy, it computes its label $D(y)$ by only including updates from *known* trustworthy nodes x: namely, $D(y) := \min\{D(x) + c_{xy}\colon (x, y) \in A$ and $(x \in N_j$ trustworthy or $x \in N - N_j)\}$. Notice that since $AL(x)$ is communicated from x to $y \in A(x)$, node y can verify whether $x \in N_j$ by simply checking if $j \in AL(x)$. As in the ancestor-list algorithm, we do not allow nodes to update their ancestors; this is determined at node y by checking if $y \in AL(x)$. Ultimately all nodes of G_j will become trustworthy and the incorrect information remaining from the prior distance labels (before the increase in arc cost) will have been overwritten with current information.

To describe our proposed *loop-free algorithm*, we subscript relevant variables by their iteration number, with the increased label detected by node j occurring at iteration $k = 0$. Namely, $D_k(x)$ is the distance label of node x at iteration k and $AL_k(x)$ is the associated ancestor list of node x. Also, $TW_k \subseteq N_j$ will denote the set of trustworthy nodes at iteration k; initially node j is the only trustworthy node. The overall structure of the algorithm is as follows. At the start of iteration k, the label and ancestor-list information is passed from nodes at iteration $k-1$, and also a token is passed from nodes in TW_{k-1} to nodes of TW_k. Then each node of TW_k computes its new label and ancestor list using information received from nodes of TW_{k-1} and $\bar{N} = N - N_j$. Updates to other nodes are carried out as in the standard ancestor-list algorithm.

Pseudocode for this loop-free algorithm is presented below. Here $d(x)$ represents the shortest-path distance to node x before an increase in label is detected at node j.

```
loop-free
    $D_0(x) := d(x)$; $AL_0(x) := AL(x)$ for all $x$;
    update new $D_0(j)$; $TW_0 := \{j\}$; $k := 0$; count := 1;
    **while** (count > 0)
        count := 0; $k := k + 1$; $TW_k := \{j\}$;
        **for** all $(x, y) \in A_j$    //update trustworthy status
            **if** $x \in TW_{k-1}$ **then** add $y$ to $TW_k$;
        **end for**;
        **for** all $x$    //send information
            send $D_{k-1}(x)$, $AL_{k-1}(x)$ to all nodes $y \in A(x)$;
        **end for**;
        **for** all $y \neq s$    //compute labels
            **if** $y \in TW_k$ **then**
                $D_k(y) := \min\{D_{k-1}(x) + c_{xy}\colon x \in B(y),$
                        $y \notin AL_{k-1}(x),\ x \in TW_{k-1} \cup \bar{N}\}$;
                update $AL_k(y)$, $\mathrm{pred}_k(y)$;
            **else**
                $D_k(y) := \min\{D_{k-1}(x) + c_{xy}\colon x \in B(y),$
                        $y \notin AL_{k-1}(x)\}$;
                update $AL_k(y)$, $\mathrm{pred}_k(y)$;
            **end if**;
            **if** $D_k(y) \neq D_{k-1}(y)$ or
                    $\mathrm{pred}_k(y) \neq \mathrm{pred}_{k-1}(y)$ **then**
                count := count + 1;
            **end if**;
        **end for**;
    **end while**
```

The final stage of this new protocol, initiated by node j, involves a mechanism for nodes to stop forwarding the special message $[j, token]$. Once all nodes of G_j that are incident to and incident from a node $y \in N_j$ have been revealed to be trustworthy, then node y can remove its trustworthy designation and return to executing the standard protocol (ancestor-list algorithm).

Before establishing the validity and complexity of the loop-free algorithm, we state the following fact, which is easily verified.

Lemma 3. *Node $y \in TW_k$ if and only if $y \in G_j$ is reachable from j by a path in G_j of cardinality (number of arcs) at most k.*

Using Lemma 3 and the definition of $D_k(y)$ in the loop-free algorithm, an inductive argument gives the following result.

Lemma 4. *Suppose $y \in TW_k$. Then $D_k(y)$ is the cost of a path P from node s to node y where all intermediate nodes of P are in $TW_{k-1} \cup \bar{N}$ and P has at most k arcs in G_j.*

Theorem 2. *The loop-free algorithm converges in at most $2n$ iterations to the correct shortest-path distances after an increase occurs to c_{ij}, resulting in $O(mn)$ worst-case complexity.*

Proof. By Lemma 4 each node $y \in TW_k$ has a feasible label $D_k(y)$, representing the cost of some path from s to y in G. By Lemma 3 all nodes of T_j become trustworthy by iteration n, since $|N_j| \le n$. At this point all nodes in the network have feasible labels, upper bounds on their true shortest-path distance from s, and so convergence is assured in at most n additional iterations. As in the ancestor-list algorithm, all data structures in the loop-free algorithm can be updated in constant time, so there is $O(m)$ work per iteration and $O(mn)$ work overall. □

To show that the loop-free algorithm does not produce cycles, let $S_k = TW_k \cup \bar{N} \subseteq N$. We can then show the following lemma.

Lemma 5. *Suppose (x, y) is an arc of the predecessor graph with $x, y \in S_k$. Then $D_k(x) + c_{xy} \le D_k(y)$.*

Proof. The increase in arc cost c_{ij} affects only the distance labels of nodes in the subtree T_j. Consequently, if $x \in \bar{N}$ then $D_k(x)$ remains constant: namely, $D_k(x) = d(x)$ for all $k \ge 0$. Also, by Lemma 4, once a node becomes trustworthy, its label is monotone nonincreasing: i.e., if $x \in TW_r$ then $D_{r+1}(x) \le D_r(x)$.

Now consider a node $y \in S_k$ that has just been updated from $x \in S_k$, so (x, y) appears in the predecessor graph at iteration k.

(a) Suppose $y \in TW_k$. Then $x \in TW_{k-1}$ or $x \in \bar{N}$; in either case $D_k(x) \le D_{k-1}(x)$ so $D_k(y) = D_{k-1}(x) + c_{xy} \ge D_k(x) + c_{xy}$.

(b) Suppose $y \in \bar{N}$. Then updates to y can be confined to those resulting from $x \in \bar{N}$. Thus $D_k(x) = d(x)$ and $D_k(y) = d(y)$ hold for all k, so we get $D_k(x) + c_{xy} = D_k(y)$.

In any case the desired result holds. □

Given this lemma, we can now show that the loop-free algorithm is indeed cycle-free, assuming that all cycles in G have positive cost.

Theorem 3. *The loop-free algorithm is cycle-free.*

Proof. Suppose to the contrary that a cycle $H = \{v_0, v_1, \ldots, v_r, v_0\}$ is formed in the predecessor graph at some iteration k. We claim that $H \cap S_k \ne \varnothing$. Otherwise, the cycle H would be formed in the original graph G before the increase in cost c_{ij}, and this can occur only if H has cost $c(H) = 0$. This contradicts our assumption that all cycles of G have positive cost. As a result, $H \subseteq S_k$ holds since untrustworthy nodes cannot update trustworthy nodes or nodes of $\bar{N}$.

By Lemma 5, $D_k(x) + c_{xy} \le D_k(y)$ holds for all $(x, y) \in H$. Hence $c(H) = \sum_{i=0}^{r-1} c_{v_i, v_{i+1}} + c_{v_r, v_0} \le (D_k(v_1) - D_k(v_0)) + (D_k(v_2) - D_k(v_1)) + \cdots + (D_k(v_r) - D_k(v_{r-1})) + (D_k(v_0) - D_k(v_r)) = 0$. Thus H has zero cost, contradicting the assumption that all cycles of G have positive cost. Hence, the stated result follows. □

Many other approaches to ensure a loop-free routing pattern have been proposed, such as Shin and Chou (1993), which maintains information on the number of minimum-cost paths as well as three routing tables. However, their procedure is not polynomially bounded. The approaches of Garcia-Luna-Aceves (1988, 1993, 1997) ensure loop-free operation, using a more complex method to screen the neighbors of a given node based on various "feasibility conditions." The approach described in Cicerone et al. (2003), which applies to undirected networks with positive arc costs, also concentrates on limiting the immediate updating of nodes within the subtree T_j. Those authors describe a fairly complex protocol, which consists of three separate phases (coloring, identifying boundary nodes, and recalculation) and focuses on minimizing the algorithm's message complexity. By contrast, our simpler approach employs the ancestor-list structure and addresses the difficulties introduced by synchronous operation, namely the creation of cycles.

5. Computational Results

To illustrate the behavior of the ancestor-list and loop-free algorithms, we compare the performances of the distributed Bellman-Ford algorithm, the predecessor algorithm, the ancestor-list algorithm, and the loop-free algorithm on four test networks, displayed in Figures 5–8. The last three of these networks exemplify network architectures commonly encountered in parallel processing (butterfly and mesh). All test networks happen to be undirected, though our procedures work equally well on directed networks. The test networks were chosen for their general applicability and their ability to exhibit a high degree of looping behavior. In each of these networks, the initial shortest-path tree from source node $s = 1$ is highlighted. Also, each unlabeled arc is assumed to have unit cost.

Two metrics are presented to compare the empirical complexity of each algorithm once a shortest-path tree T has been initially established and then an increase in the cost c_{ij} occurs for a single arc $(i, j) \in T$. First, we keep track of the number of iterations required for convergence, referred to in the literature as the *time complexity* of the algorithm. In other words,

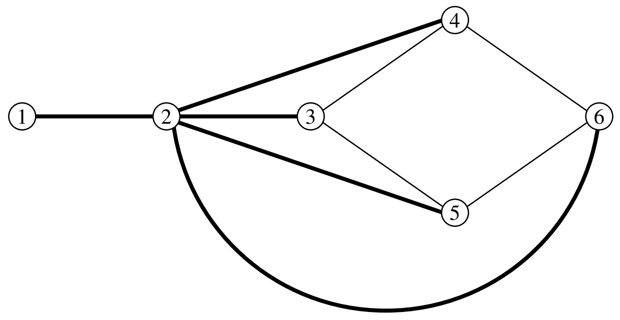

Figure 5 Network *A*

the time complexity is the smallest integer h such that $D^{h-1}(v) = D^h(v)$ holds for all nodes v. Second, we record the number of times that nodes update their distance labels. Since each update triggers the sending of a data message, the number of updates measures the *message complexity* of each algorithm.

For each of these networks we increase the cost of a certain arc $(i, j) \in T$ by a prescribed amount and tabulate the time and message complexity incurred by each of the four algorithms. The selected arcs and the cost changes are noted in Columns 2–3 of Table 1. The first column gives the test network, its number of nodes n, the depth Δ of the shortest-path tree T, and the size of the subtree T_j of T rooted at node j. The number of iterations (*itns*) and updates (*updts*) required until convergence are then listed for each of the four algorithms. All algorithms were written using ANCI C++ and executed on a Dell Optiplex (2.4 GHz processor, 1 G memory) running Windows XP Service Pack 1.

As seen in Table 1, networks A, C, and D provide challenging instances for the Bellman-Ford and predecessor algorithms. In each of these cases, once the increase in arc cost is detected, most of the updates occur within T_j and there are no opportunities for

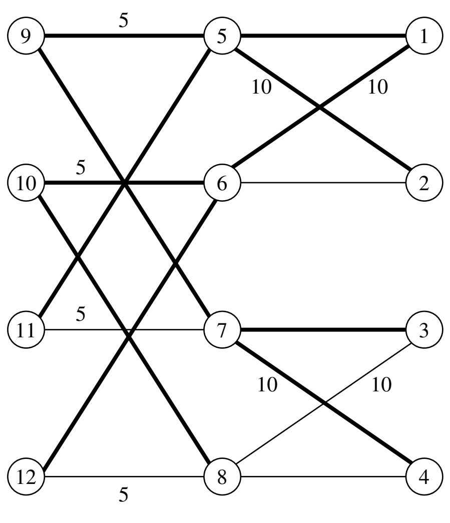

Figure 6 Network *B* (Butterfly)

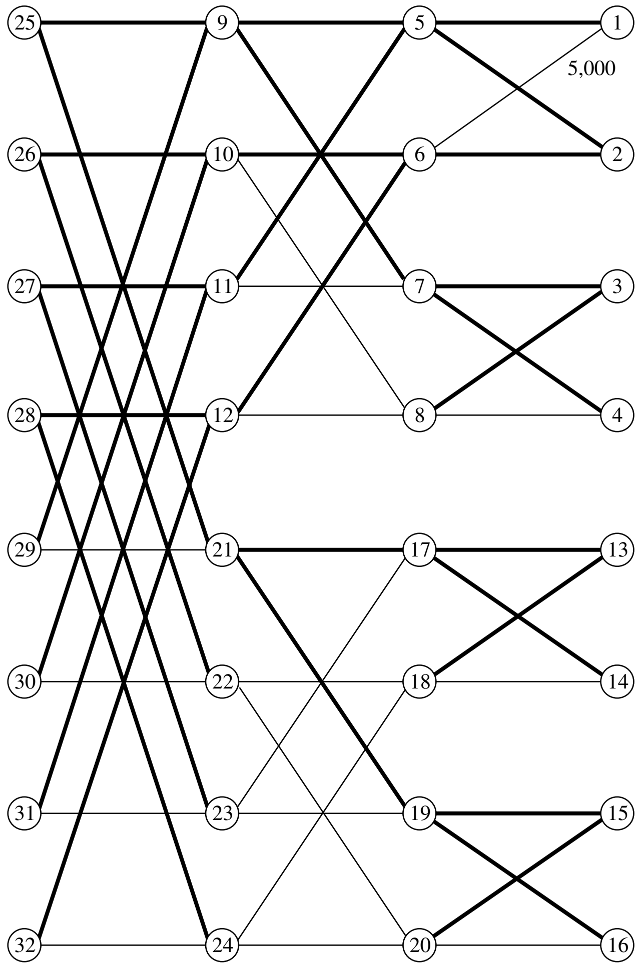

Figure 7 Network *C* (Butterfly)

nodes outside of T_j to update nodes within T_j. This exacerbates the looping problem for these algorithms. In network B there are several arcs incident with the source node $s = 1$ of the shortest-path tree T; here the Bellman-Ford and predecessor algorithms converge to optimality in a reasonable number of iterations and updates. In general, Table 1 indicates that the ancestor-list and loop-free algorithms exhibit superior performance, with the loop-free algorithm typically dominating all algorithms in terms of operation counts.

We have also collected CPU times by running each code 1,000 times on each test case; the average times (in milliseconds) are reported in Table 2 for all instances in which a nonzero running time could be detected. The results in Table 2 show that the ancestor-list and loop-free algorithms, which were implemented using bit vector operations (Aho et al. 1974) to update the ancestor lists, achieve improved running times compared to the original and predecessor algorithms, especially when opportunities for looping exist. The additional overhead required by the predecessor algorithm in networks A, C, and D shows

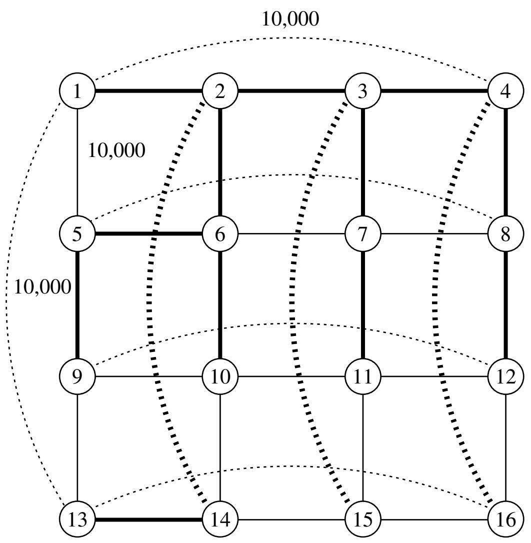

Figure 8 **Network *D* (Mesh)**

up as increased CPU times compared to the original Bellman-Ford algorithm. On the other hand, the additional work required by the loop-free and ancestor-list algorithms to update $AL(y)$ is insignificant compared to the savings achieved, especially in networks A, C, and D. Overall, the loop-free algorithm is comparable to or slightly faster than the ancestor-list algorithm in running time, despite a more complicated protocol.

As the theoretical results (Theorems 1 and 2) suggest, the ancestor-list and loop-free algorithms converge in $O(n)$ iterations to the correct distance labels following an increase in arc cost, while the Bellman-Ford and predecessor algorithms are assured only of a pseudopolynomial time bound on convergence. The results of Tables 1 and 2 clearly show the dependence on the arc cost c_{ij} for these latter two algorithms. Empirically, the ancestor-list algorithm converges within n iterations (compared to the $3n$ bound from Theorem 1). The loop-free algorithm empirically converges in many fewer iterations than the $2n$ bound available from Theorem 2. Indeed, the loop-free algorithm converges here within $\Delta + 1$ iterations. Moreover, in each of the test problems the message complexity for the loop-free algorithm is at most $|T_j| + 1$.

Table 2 **Comparison of CPU Times**

Network	(i, j)	c_{ij}	Bellman-Ford	Predecessor	Ancestor	Loop-free
A	(1, 2)	1,000	0.28	0.32	0.00	0.01
B	(1, 5)	7	0.01	0.01	0.01	0.02
		15	0.02	0.01	0.01	0.02
		50	0.02	0.01	0.01	0.02
		100	0.02	0.01	0.01	0.02
B	(1, 6)	50	0.01	0.01	0.01	0.01
		100	0.01	0.01	0.01	0.01
		1,000	0.01	0.01	0.01	0.01
C	(1, 5)	3,000	4.73	5.55	0.07	0.06
		4,000	6.31	7.37	0.08	0.06
		5,000	7.86	9.29	0.09	0.07
D	(1, 2)	5,000	4.58	4.63	0.03	0.02
		7,500	6.90	6.96	0.03	0.02
		10,000	9.19	9.29	0.04	0.02

6. Conclusions

The Bellman-Ford algorithm can be implemented as a distributed algorithm in a straightforward way, and in this form it provides a simple approach for routing messages in a communication network with stable link costs. However, its performance can notice-

Table 1 **Comparison of Operation Counts**

			Bellman-Ford		Predecessor		Ancestor		Loop-free	
Network	(i, j)	c_{ij}	itns	updts	itns	updts	itns	updts	itns	updts
A $(n = 6, \Delta = 2, \lvert T_j\rvert = 5)$	(1, 2)	5	6	19	6	19	6	13	3	5
		10	11	44	11	44	6	13	3	5
		100	101	494	101	494	6	13	3	5
		1,000	1,001	4,994	1,001	4,994	6	13	3	5
B $(n = 12, \Delta = 4, \lvert T_j\rvert = 7)$	(1,5)	7	10	18	5	7	5	7	5	7
		15	18	38	9	17	5	8	5	7
		50	23	52	10	18	6	11	5	7
		100	23	52	10	18	6	11	5	7
B $(n = 12, \Delta = 4, \lvert T_j\rvert = 4)$	(1,6)	50	4	4	4	4	4	4	5	5
		100	4	4	4	4	4	4	5	5
		1,000	4	4	4	4	4	4	5	5
C $(n = 32, \Delta = 7, \lvert T_j\rvert = 31)$	(1, 5)	3,000	3,006	46,500	3,006	46,479	24	183	8	31
		4,000	4,006	62,000	4,006	61,997	24	183	8	31
		5,000	5,006	77,488	5,006	77,486	24	183	8	31
D $(n = 16, \Delta = 5, \lvert T_j\rvert = 15)$	(1, 2)	5,000	5,004	37,500	5,004	37,500	16	59	6	15
		7,500	7,504	56,250	7,504	56,250	16	59	6	15
		10,000	10,003	74,994	10,003	74,994	16	59	6	15

ably degrade when cost changes (in particular, cost increases) occur on network links. We have presented two new approaches, the ancestor-list algorithm and the loop-free algorithm, which require maintaining some additional information compared to the distributed Bellman-Ford algorithm. As a result, we can show that these new algorithms achieve time bounds of $3n$ and $2n$ iterations, improving on the pseudopolynomial time bound for the distributed Bellman-Ford algorithm and maintaining the $O(mn)$ worst-case complexity of the original Bellman-Ford algorithm. Our protocols and proofs are fairly simple, especially in comparison to those available for alternative algorithms proposed in the literature. Preliminary computational studies show that these new algorithms in practice behave even better than our theoretical bounds suggest. Future research would be directed to analyzing why their actual performance exceeds our expectations.

Acknowledgments
A preliminary version of this paper appeared in the *Proceedings of the INOC 2005 Conference*, Lisbon, March 20–23, 2005, pp. 449–454. The authors thank the referees for the many constructive comments, which have helped to improve this paper.

Appendix
Although in practice this seems to occur rarely, it is nevertheless possible for I_k to be nonempty at some iteration k of the ancestor-list algorithm. To illustrate such a situation, consider Figure A1. Assume that at iteration $i-1$ node a receives an update, increasing its label $D(a)$. At iteration i, node a has updated node b using this increased label. At iteration $i+1$, node b has passed its increased node label to c, and as a result c gets updated instead by f. Note that $a, b \notin AL_{i+1}(c)$, but rather $d, e, f \in AL_{i+1}(c)$. Further, suppose that at this iteration node d receives an update, increasing its distance label. At iteration $i+2$, node c has updated b, giving $c, d, e, f \in AL_{i+2}(b)$. Further, d has sent its updated information to e, and e is updated now by a so that $a \in AL_{i+2}(e)$. Notice that node e cannot update node d at iteration $i+2$ because $d \in AL_{i+1}(e)$. Finally, at iteration $i+3$, node b has updated node a so that $e \in AL_{i+3}(a)$, and node e has updated nodes d and f. Because $e \in AL_{i+3}(a) \cap AL_{i+3}(d) \cap AL_{i+3}(f)$ (i.e., e is now an ancestor of all its neighbors), node e receives no valid updates so that $D(e) = \infty$ and $e \in I_{i+4}$.

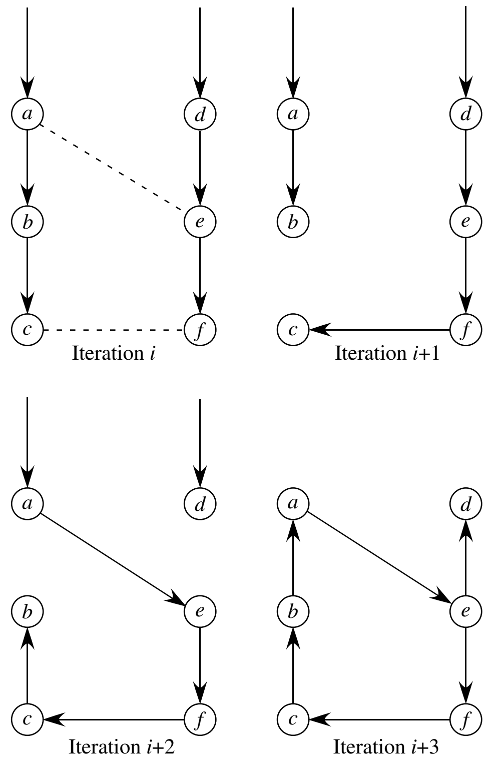

Figure A1 **Example Showing I_k Can Be Nonempty**

References
Aho, A., J. Hopcroft, J. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

Ahuja, R., T. Magnanti, J. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ.

Bellman, R. 1958. On a routing problem. *Quart. Appl. Math.* **16** 87–90.

Bertsekas, D., R. Gallager. 1987. *Data Networks*. Prentice Hall, Englewood Cliffs, NJ.

Bertsekas, D., J. Tsitsiklis. 1989. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs, NJ.

Cegrell, T. 1975. A routing procedure for the TIDAS message-switching network. *IEEE Trans. Comm.* **23** 575–585.

Cicerone, S., G. Di Stefano, D. Frigioni, U. Nanni. 2003. A fully dynamic algorithm for distributed shortest paths. *Theoret. Comput. Sci.* **297** 83–102.

Garcia-Luna-Aceves, J. 1988. A distributed, loop-free, shortest-path routing algorithm. *Proc. IEEE INFOCOM 88, New Orleans, March 1988*, IEEE Computer Society Press, New York, 1125–1137.

Garcia-Luna-Aceves, J. 1993. Loop-free routing using diffusing computations. *IEEE/ACM Trans. Networking* **1** 130–141.

Garcia-Luna-Aceves, J. 1997. A path-finding algorithm for loop-free routing. *IEEE/ACM Trans. Networking* **5** 148–160.

Humblet, P. 1991. Another adaptive distributed shortest path algorithm. *IEEE Trans. Comm.* **39** 995–1003.

Jaffe, J., F. Moss. 1982. A responsive distributed routing algorithm for computer networks. *IEEE Trans. Comm.* **30** 1758–1762.

Johnson, M. 1984. Updating routing tables after resource failure in a distributed computer network. *Networks* **14** 379–391.

Merlin, P., A. Segall. 1979. A failsafe distributed routing protocol. *IEEE Trans. Comm.* **27** 1280–1287.

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

Ramarao, K., S. Venkatesan. 1992. On finding and updating shortest paths distributively. *J. Algorithms* **13** 235–257.

Schwartz, M. 1986. *Telecommunication Networks: Protocols, Modeling, and Analysis*. Addison-Wesley, Reading, MA.

Shin, K., M.-S. Chen. 1987. Performance analysis of distributed routing strategies free of ping-pong-type looping. *IEEE Trans. Comput.* **36** 129–137.

Shin, K., C.-C. Chou. 1993. A simple distributed loop-free routing strategy for computer communication networks. *IEEE Trans. Parallel and Distrib. Syst.* **4** 1308–1319.