A LOOP-FREE EXTENDED BELLMAN-FORD ROUTING PROTOCOL WITHOUT BOUNCING EFFECT

Chunhsiang Cheng, Ralph Riley, and Srikanta P.R. Kumar*

Department of Electrical Engineering
and Computer Science
Northwestern University
Evanston, IL 60208
Tel: (312) 491-7382

J.J. Garcia-Luna-Aceves**

Network Information Systems Center
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

ABSTRACT:

Distributed algorithms for shortest-path problems are important in the context of routing in computer communication networks. We present a protocol that maintains the shortest-path routes in a dynamic topology, that is, in an environment where links and nodes can fail and recover at arbitrary times. The novelty of this protocol is that it avoids the bouncing effect and the looping problem that occur in the previous approaches of the distributed implementation of Bellman-Ford algorithm. The bouncing effect refers to the very long duration for convergence when failures happen or weights increase, and the nonterminating exchanges of messages, or counting-to-infinity behavior, in disconnected components of the network resulting from failures. The looping problems cause data packets to circulate and, thus, waste bandwidth.These undesirable effects are avoided without any increase in the overall message complexity of previous approaches required in the connected part of the network. The time complexity is better than the distributed Bellman-Ford algorithm encountering failures. The key idea in the implementation is to maintain only loop-free paths, and search for the shortest path only from this set.

* This research was sponsored by grants from Bell Northern Research, U.S. West Advanced Technologies, and Ameritech.

** This work was sponsored by SRI International IR&D funds.

1. INTRODUCTION

One of the widely used techniques for routing in communication networks is via distributed algorithms for finding shortest paths in weighted graphs [9,10,13,14]. The well known distributed Bellman-Ford (BF) algorithm (implemented initially in ARPANET [14]) is simple, and the distance and the routing-tables are easy to maintain [2]. However, this protocol has several major drawbacks. Firstly, the response of this protocol to link or node failures can be very slow. This is due to the possibility that the distances maintained, and exchanged with neighbors, in the internal distance-table or routing-table of each node, may correspond to paths with loops ("bouncing effect" [20]). Thus, nodes may engage in a prolonged exchange of such distances before converging to the shortest paths. Moreover, if the network is disconnected, the protocol is not guaranteed to terminate. (This is the so called counting-to-infinity problem, where each node keeps indefinitely increasing its distances to the unreachable destinations.) Another shortcoming of this protocol is that it is not loop-free in the following sense [3,4,9,13,17]: at any moment, the paths implied by the routing-tables of all nodes taken together can have loops (i.e., if a path to a destination is traced going from the routing-table of one node to that of another node, a node may be visited more than once before the destination is reached. If such routing-table loops persist for a long time, looping of data to be routed may occur resulting in considerable overhead. Avoiding the bouncing effect does not necessarily imply that routing-table loops are eliminated. (The looping of data packets may not be completely avoided even if the routing-tables are loop-free at all time [2].) Here, we take a protocol to be loop-free, if it does not have routing-table loops mentioned above [9].

One way to overcome the termination (or counting-to-

infinity) problem is to use some additional information, such as the size or diameter of the network [20]. However, such information may vary from time to time, if the network topology is dynamic, and in such cases the convergence of the protocol via this approach will be too slow. The new ARPANET routing protocol [2,15] runs the Dijkstra's algorithm periodically at each node based on the information of the whole network. Although the looping problem is alleviated to some extent, in a slowly varying network, the overhead, in terms of the messages and the local memory required, is too high as each node must gather the information about the whole topology.

Merlin and Segall [16] proposed a synchronization approach to achieve loop freedom. In this approach, there is additional overhead due to the cost of the synchronization phase. In addition, the speed of the convergence can be slower than the BF algorithm, when no looping is encountered.

In fact, achieving loop-freedom in the distributed BF algorithm is not difficult in networks with uniform weight on each link. Chu [5] proposed the downstream and upstream idea to avoid loops in minimum hop routing. A similar idea was adopted by Shin and Chen [19] for nonuniformly weighted networks to avoid two-node looping. This algorithm can also be extended to a kth order algorithm which avoids all loops with no more than k hops. However, in this case, the size of the control messages and the local memory required grow proportional to k.

Jaffe and Moss [13] used the freezing technique to delay the response of a node at the moment the node loses its preferred neighbor to prevent the possibility of looping. Garcia-Luna-Aceves [9] extended the same idea to achieve a lower message complexity, and also presented a formal proof. For a further critique of the previous approaches, see Garcia-Luna-Aceves [9,10].

In this paper, we present a shortest path routing protocol, in which each node maintains information about some simple paths in its local memory. Note that knowing the entire path is subtantial for determining whether a path is simple or non-simple. However, it is possible to obtain the entire path to every destination by simply knowing the node next to each destination, thus eliminating the overhead caused by installing the list of all the nodes in the path to any destination (the approach adopted by Shin and Chen [19]). A mechanism to achieve this was first suggested by Hagouel [11], who attempted to do the source routing using an algorithm almost identical to the Bellman-Ford algorithm. Garcia-Luna-Aceves adopted this mechanism to attempt to reduce the looping problem of the Bellman-Ford algorithm for min-hop routing in packet-radio networks [8]. A back-tracking technique was applied to maintain the routing table, which may cause exponential complexity of nodal computation time. Humblet proposed a breadth first search technique to conquer this drawback [12]. A more efficient algorithm, in terms of lower nodal computation time complexity, less computation storage, and smaller message size, is presented in this paper independently of the work in [12]. The update of the distance vectors are sent only to selected neighbors, so as to maintain distances of only simple paths, and consequently avoiding the bouncing effect (and the counting-to-infinity problem) and converging to the correct distances quickly. Moreover, the protocol is simple and the size of the local memory required and the message size are increased by a factor of only logN bits (assuming logN bits for a node identification) the requirements of the original distributed BF algorithm. On top of this basic protocol, loop-freedom can be achieved through inter-neighbor coordination [10]. A technique is also presented to prove the correctness of a protocol that can be embedded in another protocol that has been proven correct.

The rest of this paper is organized as follows. Section 2 presents the network model assumed in the basic protocol, which is described in Section 3. Section 4 presents the correctness proof of the protocol and Section 5 describes the additions needed to the basic protocol to achieve the loop-freedom property. Performance issues are discussed in Section 6, and Section 7 presents conclusions.

2. NETWORK MODEL AND NOTATIONS.

2.1 Network Model.

The environment for the protocol is an asynchrounous point-to-point network presented by an undirected weighted graph G(V,L), where V is the set of nodes numbered 1,2,..,N, and $E \subseteq VxV$ is the set of links. Each node is a computing unit involving a processor, a local memory, and also an input queue and an output queue with unlimited capacity. Each functional link (i,j), assigned with a weight $d_{ij} > 0$, is a bidirectional communication line connecting nodes u and v. Each node knows only its local environment (the numbers of the neighbors and the weights of the adjacent links) and follows the same protocol consisting of sending and receiving messages over the adjacent links, and processing these messages. The received (sent) messages are

put in the input (output) queue on a first-come-first-served basis, and are processed in that order.

A communication link in a dynamic network has the following properties. Messages can only be sent and received over a link which is functioning. However, a message sent need not arrive at the receiver, as the link may fail during transmission. When the link is functioning, messages can be independently transmitted in both directions, and they arrive at the other end node after a finite undetermined delay, without error and in sequence. Whenever a link fails or recovers, each end node is notified in a finite time, but not necessarily at the same time. When a link recovers, there are no messages in transit through it, nor are there messages waiting to be sent over it (i.e., all messages sent out for transmission on link are deleted after the link fails). A node failure/recovery is taken to be the failure/recovery of all adjacent links. A change in the weight of a functioning link is also assumed to be notified to both the end nodes in a finite time. These services are assumed to be provided by a lower level (link) protocol. The assumptions stated above are standards [1,18].

2.2 Notations and Definitions.

A *path* (*route*) from node i to node j, denoted R_{ij}, is a sequence of nodes $R_{ij}=(i,n_1,n_2,..,n_r,j)$ where (i,n_1), (n_r,j), and (n_x,n_{x+1}) for $1 \le x \le r-1$ are links. A path from i to j via node k, a neighbor of i, is denoted R_{ij}^k. The distance of a path is the sum of the weights of all the edges in that path.

A *simple path* (also an elementary path [6]) from i to j is a sequence of nodes with no node being repeated more than once. The paths between any pair of nodes, and the distances, change over time in a dynamic network.

At any point in time, a node i is said to be connected to node j iff there exists at least one path between these two nodes in the graph at that time. The network, at any time, is said to connected iff every pair of (functioning) nodes are connected at that time.

The *head* of a path R_{ij} is defined to be the last node preceding node j in the sequence of nodes in R_{ij} (i.e., if $R_{ij}=(i,n_1,n_2,..,n_r,j)$, then head of R_{ij} is n_r if $r > 0$, and equal to i if $r = 0$).

3. A BASIC SHORTEST PATH PROTOCOL TO AVOID BOUNCING EFFECT

In this section, we present a refinement to the distributed BF algorithm to avoid the bouncing effect and the counting-to-infinity problem. Additional modifications needed to achieve loop-freedom (no routing-table loops) will be discussed in Section 5. Before we outline the refinements, the original distributed BF algorithm is briefly reviewed.

3.1 Distributed Bellman-Ford Algorithm

In the asynchrounous distributed BF algorithm (which falls into the class of distance vector algorithms [9]), the nodes asynchrounously exchange their routing vectors representing shortest path distances, computed according to the BF iteration[1]. Each entry in a routing vector is a message which contains a destination node and a distance to that destination from the node sending the vector. When a node receives a routing vector from a neighboring node, it updates its distances to other nodes via this neighbor, and any changes in the routing table (current shortest paths) are then sent to all neighbors. Let $D_i = [D_{ij}^k]$ denote the distance matrix (with rows corresponding to destinations and columns to neighbors) stored in node i, with entry D_{ij}^k representing the distance from i to j via neighbor k of i. The routing table is an array (derived from D_i) with one entry for each destination. Each entry is a triplet specifying the destination (say j), the preferred neighbor P_{ij} (P_{ij} is the neighbor along the shortest path R_{ij} to the destination), and the current shortest distance (minimum of row j in D_i), denoted by $RDIST_i(j)$. The BF algorithm converges to the shortest distances for all connected pairs of nodes.

The basic steps of a version of the distributed BF algorithm is as shown as Algorithm 1 [2,14,20,21].

The notification link failure/recovery and the weight changes are presented to a node by its lower level (link) protocol. For link weight changes, say of link (i,k), the response is as if a (N-1)-entry vector is received on link (i,k), where each entry corresponds to $(k,D_{ij}^k - d_{ik}^*)$ for all j (d_{ik}^* is the old link weight of link (i,k)).

The bouncing effect refers to the behavior in which a node u will keep on increasing the distance to a destination through some neighbors who, in fact, do not have the path corresponding to such distance toward the destination without going through u. This immediately results in a longer duration for the nodes to update their distance and routing tables to be correct, and, consequently, induces

[1] Bellman-Ford iteration equation for asynchrounous distributed model:

$D_{ij} := \min \{d_{ik} + D_{kj} \mid \text{for all neighbors k of i}\}$, $D_{ii}:=0$ for all n_i

Algorithm 1: Distributed Bellman-Ford

```
Response of node i:
  Upon receiving vector V^k on link (i,k).
    (0)  begin
           V^i ← Ø
    (1)    for each entry (j,D_kj) in V^k do
         (* V^k is a set of 2-tuples *)
           begin
             D^k_ij ← D_kj + d_ik
           (* copy into column j *)
           end
    (2)    for each row j in D_i do
           begin
             if min_k D^k_ij ≠ RDIST_i(j) then
             begin
               RDIST_i(j) ← min_k D^k_ij
               (* update routing table *)
               P_ij ← arg min_k D^k_ij
               (* set preferred neighbor *)
               V^i ← V^i ∪ { (j, RDIST_i(j)) }
             end
           end
    (3)    if V^i ≠ Ø then
           begin
             send V^i to all neighbors.
             (* send changes *)
           end
         end
  Upon receiving notification Failure(i,k)
  (* link (i,k) is not functioning anymore *)
    (4)  begin
           delete column k in D_i and execute steps 2 and 3.
         end
  Upon receiving notification Recover(i,k,d_ik)
  (* link (i,k) is now functioning *)
    (5)  begin
           insert column k in D_i.
           respond as if a single entry in V^k = {(k,d_ik)}
           is received on link (i,k).
           copy whole routing table into V^i and send it to k.
         end
```

higher time complexity.

The following example, illustrated by Fig. 1, summarizes the looping and bouncing effect scenarios.

Consider destination node n_1. Assume that, each node obtains the correct distance and routing tables by running the BF algorithm. It is easy to see that nodes n_2 and n_3 will choose nodes n_1 and n_2 as their preferred neighbors respectively. If link (n_1,n_2) fails, node n_2 will choose node n_3 as the preferred neighbor based on its distance table. Thus, a routing-table loop occurs between nodes n_2 and n_3. Furthermore, nodes n_2 and n_3 have the distances 3 and 2,

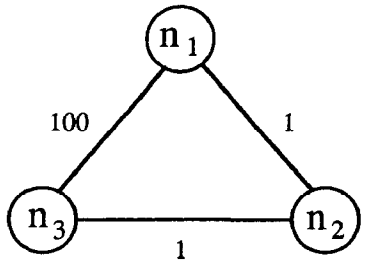

FIG. 1 -- A network topology that will result bouncing effect and routing-table loop when link (n_1,n_2) fails, and counting-to-infinity when (n_1,n_3) also fails

which are much less than 100 to destination n_1. Due to the BF algorithm, node n_3 will update its distance to be 4, which again will cause node n_2 increase its distance to be 5. Clearly, nodes n_2 and n_3 will keep on increasing their distances until node n_3 reaches a distance of 102 to n_1, and chooses n_1 as its preferred neighbor. After this, the distance converges. Thus, we observe that, with the BF algorithm, nodes n_2 and n_3 engage in a prolonged process of message exchange until n_3 reaches a distance to n_1 through neighbor n_2 greater than 100. Only after that moment, n_3 switches to the correct preferred neighbor and has the correct distance. This scenario represents the bouncing effect. This problem is much worse if link (n_1,n_3) also fails later; which is equivalent to link (n_1,n_3) having an infinity weight. Apparently, nodes n_2 and n_3 will keep on increasing their distance to n_1 without bounds. This is the so called counting-to-infinity problem arising from network partitions. Note that the counting-to-infinity problem can be viewed as a special case of the bouncing effect.

3.2 A Basic Protocol (Extended Distributed BF)

In this section, we present a refined shortest-path protocol that avoids the bouncing effect (the slow response to link failures or link weight increase) and the counting-to-infinity behavior. We start by providing the motivation and some intuition behind these refinements.

It is possible to eliminate the counting-to-infinity behavior in a straightforward way if an upper bound on the distance between any pair of nodes (in the dynamic network) can be known in advance. A node encountering a distance greater than this bound to a destination can immediately mark that destination as disconnected [20]. The disadvantage of this approach is that it requires a prior and precise knowledge of such an upper bound. More importantly, this mechanism has no influence on the convergence behavior in

connected components of the network; thus, the bouncing effect which accounts for is not eliminated.

As can be seen from the example in the previous section, the bouncing effect and the counting-to-infinity behavior arrise due to the fact that a node may offer its neighbor a distance corresponding to a path that has that neighbor as an internal node of that path. Thus, once the distance is updated, the distance may correspond to a non-simple path. To explain this remark further, one can observe from Algorithm 1 that, at any moment, any finite distance maintained in the distance table or the rouitng table of a node (say i) for a destination (say j) is generated over the network by sequentially accumulating a set of link weights, which either existed at some previous moment or currently exist in the network. In addition, this set of links form a path or a route from node i to node j. For example, a distance entry D in the distance or routing table of node i, maintained at time t, can be expressed as $D=\sum_{m=1}^{r} d_{n_m n_{m+1}}(t_m)$where $n_1=i$, $n_{r+1}=j$, and $d_{n_m n_{m+1}}(t_m)$ is the weight of link (n_m,n_{m+1}) existing in the network at some time $t_m \leq t$. Indeed, node i gets informed about this distance D via the sequence of nodes j, n_r, ..., n_2, i. (In other words, node n_m is informed of the distance $D=\sum_{k=m}^{r} d_{n_k n_{k+1}}(t_k)$ to destination j at time t_m, when it transmits via the routing vector to node n_{m-1}). We shall refer to this path $(n_1=i, n_2, ..., n_r, j)$ as the *path implicit* (or *path corresponding*) to this distance D. Note that this path is derived from the history of the link weights in the dynamic network, and not from the network topology at any specific moment. The bouncing effect and the counting-to-infinity behavior in Algorithm 1 are due to the fact that the path implicit in a distance entry in a nodal routing table may be non-simple. For instance, in the example of the previous section, the distances maintained at node n_2 to destination n_1 through neighbors n_1 and n_3 is 1 and 3, which imply the paths n_2n_1 and $n_2n_3n_2n_1$, respectively. The bouncing effect arises due to the fact that node n_3 offers its neighbor n_2 distance 2 corresponding to a path $n_3n_2n_1$ which, has n_2 as an internal node of that path. Thus, once the distance table is updated due to the failure of link (n_1,n_2), the distance 3, maintained by node n_2, implies the non-simple path $n_2n_3n_2n_1$, and will be used as the current shortest distance and passed to node n_3. This, consequently, results the prolonged message exchange.

The bouncing effect can be avoided if a protocol can be designed that searches for the shortest paths among only simple paths. Such a design can be accomplished in a relatively straightforward way by modifying Algorithm 1 so that each node, in addition to storing the distances, also stores the implicit paths for these distances in its distance table and routing table. These paths can be easily generated. A node i, receiving a distance D_{kj} and path R_{kj} from neighbor k, will record in its distance table the distance $D_{ij}^k=D_{kj}+d_{ik}$, and path R_{ij}^k as the path R_{kj} augumented by node i. Thus, a node can determine whether or not the path implicit in a distance is simple, and ignore all non-simple paths. Non-simple paths can also be avoided by having each node send a distance entry in its routing table (in step (3) of Algorithm 1) to a neighbor only if that neighbor is not in the path implicit in this distance. The proof of correctness of this modification can be established along the lines of the analysis of Algorithm 2 presented later; see also Shin and Chen [19], who adopt a similar approach. However, the overhead in this approach is large, because an entire path corresponding to each distance entry is recorded; consequently, the message size and the local storage for each node is O(N) times that required for Algorithm 1.

A main feature of the protocol presented here (Algorithm 2) is the use of the notion of *head of path* , which permits each node to infer the path implicit in a distance entry without adding excessive overhead to update messages or local storage. Each entry in the distance and routing table of a node is associated with a head of the path corresponding to this distance entry. Suppose the design of the protocol is such that, at all times, the distance and routing table at each node satisfies the following property (a form of local shortest path consistency): the path implicit in a distance entry D\a\al(k,ij) with associated head h\a\al(k,ij)=h is the path implicit in D_{ih}^k argumented by node j (i.e., from node h to node j). If each column of the distance-table (and also the routing-table) satisfies this property, then the path implicit in each distance entry can be inferred from the head of the path information above, which in turn can be used to maintain only simple paths. The advantage of this approach is that the additional overhead (over Algorithm 1) is only one node identity per table entry (which is much less than the overhead incurred in [19]). However, this mechanism is best suited for the context of an all-pairs shortest path protocol (i.e., when all nodes wish to find their shortest paths to all other nodes).

Protocol Description

We take the routing table of a node (say i) to be a vector with each entry a quadruple specifying the destination (say j), the preferred neighbor P_{ij}, the current shortest distance ($RDIST_i(j)$), and the head $HEAD_i(j)$ of the path R_{ij}.

The distance table (denoted by DT_i) is a matrix with an entry being a pair (D_{ij}^k, h_{ij}^k) where D_{ij}^k represents the distance from i to j via k, and h_{ij}^k represents the head node of the corresponding path. Infinite distances are denoted by ∞ and null head of paths by *. The rows in the distance table correspond to destinations, and columns to neighbors. (As before, the size of the distance-table and routing vector depends on the number of functioning nodes and links.)

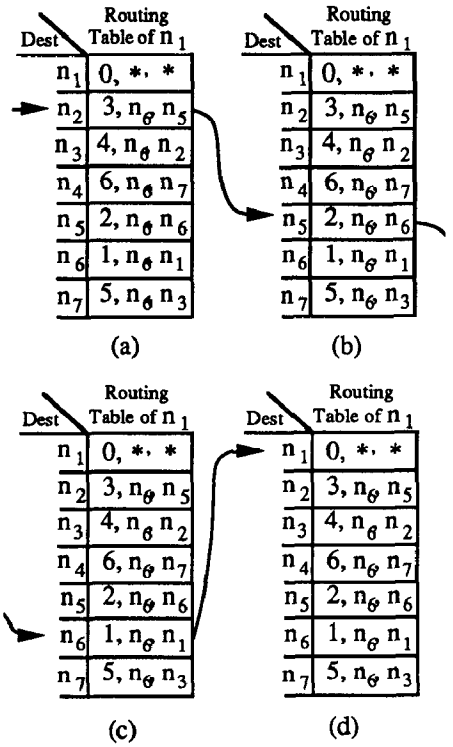

FIG 2 -- An example showing how node n_1 can determine whether a neighbor (n_7) is in its path to n_2 by tracing the head of paths in the routing table of n_1. The destinations are in the leftmost column and the other three (x,y,z) correspond to the distance, preferred neighbor, and head of path, respectively.

Fig.2 illustrates how a node determines if its neighbor is in the path from it to a destination. Suppose that n_1 wants to determine (by tracing head of paths) if its neighbor n_7 is in its shortest path to destination n_2. Initially, n_1 starts the trace from the destination (Fig. 2 (a)) and finds the head, n_5 (Fig. 2 (b)), of its shortest path to n_2. Subsequently, n_1 finds the head of its path, n_6, to n_5 (Fig. 2 (c)). Finally, n_1 finds the head of its path to n_6 to be n_1 (Fig. 2 (d)). Thus, upon reaching itself in the trace (Fig. 2 (d)), n_1 determines that n_7 is not in the set of head of the paths encountered during the trace. Equivalently, node n_7 is not in the path from n_1 to n_2. If n_7 is in the path from n_1 to n_2, then at some step of tracing, n_7 is encountered, and, the tracing can terminate at that step. The sequence of the nodes that appear in searching the path from n_1 to n_2 by tracing from n_2 back to n_1, represents a path. The path derived using the head information from the node's distance table or routing table, as the node's distance (D) to a destination, is called the *path extracted* from D. For instance, the path extracted from distance 3 to node n_2 is $n_1n_6n_5n_2$ in Fig. 2.

Note that checking if a neighbor is in the path extracted from a shortest path distance by the sender is done by assuming that such a path extracted from routing table using the head information is the path implicit in this shortest path distance. However, if we simply apply step (2) in Algorithm 1 to generated the routing table, then the resulting routing table may not have the desired property that the path extracted from a distance in the routing table is the path implicit in this distance. This can be seen from the following example. Suppose that the distance table of node i has the path abci implicit in distance 3 and path bci implicit in distance 2 through neighbor c. Moreover, b is neighbor of node i and d_{ib} is 1. If distance 3 is the minimum among row a, and distance 1 is the minimum among row b, then, based on step (2) of BF algorithm, we can have the path abi, different to adci, implied by distance 3 to go to node a in i's routing table. Clearly, node i will send this distance to neighbor c, which violates the required property. To conquer this, we modify the rule to generate the routing table, such that, the distance in the routing table to go to any node j determined by choosing from column k iff D_{ij}^k is the minimum among row j of distance-table and each node v in the path extracted from D_{ij}^k, which is assumed to be the same as the path implicit in D_{ij}^k, must be that D_{iv}^k is also the minimum among row v and is chosen to be put in the routing vector.

The two main differences between the basic protocol and Algorithm 1 are the function used to check if a neighbor is in the extracted paths when sending updated shortest path distance, and the procedure used to update the routing table. These two are summarized in function IN_PATH and procedure RT_UPDATE subsequently in this section. For the neighbors that are in the path to a destination (say j), an entry $(j,\infty,*)$ is sent as indicated in Step 3 of Algorithm 2. The other steps in this algorithm, updating the distance table (Step 1) and in response to receiving a routing vector from a message from a neighbor, are essentially the same as in Algorithm 1 (except for the obvious modifications necessary for updating the head argument in each entry). Similarly, the responding to failures and recoveries require only a simple modification.

Note that, for convenience, weight change on any link (i,j) is treated as if (i,j) fails and immediately recovers with the new weight. In addition, failure or recovery of a node is treated as if all the link adjacent to that node fail or recover. Therefore, the events that the protocol can encounter are link failures and link recoveries.

Algorithm 2: A Basic Protocol

Response of node i:

Upon receiving vector $V^{k,i}$ on link (i,k).
(* V^k is a set of triples *)

(0) **begin**
$V^i \leftarrow \emptyset$; $V^{i,b} \leftarrow \emptyset$ for all neighbors b

(1) **for** each triple $(j,D_{kj},h_k(j))$ in $V^{k,i}$, $j \neq i$ **do**
begin
$D^k_{ij} \leftarrow D_{kj} + d_{ik}$; $h^k_{ij} \leftarrow h_k(j)$
(* copy into column j *)
end

(2) if there are b and j such that $D^b_{ij} < D_{ij}$ or $k = P_{ij}$
(* routing table has to be changed *)
then update routing table
(* **call procedure RT_UPDATE** *)
else $V^i \leftarrow \emptyset$

(3) **if** $V^i \neq \emptyset$ **then**
begin
for each neighbor b **do**
(* send changes *)
begin
for each triplet $t=(j, RDIST_i(j), HEAD_i(j))$ in V^i
do
(* send updates to select neighbors and infinity to others *)
begin
if b is in the path from i to j in the routing table
(* **call function IN_PATH as defined below** *)
then $V^{i,b} \leftarrow V^{i,b} \cup \{(j,\infty,*)\}$
else $V^{i,b} \leftarrow V^{i,b} \cup t$
end
send $V^{i,b}$ to neighbor b.
end
end
end

Upon receiving notification Failure(i,k)
(* link (i,k) not functioning anymore *)

(4) **begin**
delete column k in D_i
execute steps 2 and 3.
end

Upon receiving notification Recover(i,k,d_{ik})
(* link (i,k) now functioning *)

(5) **begin**
insert column k in D_i
respond as if a single entry in $V^{k,i} = \{(k,d_{ik},i)\}$
(* i, is $HEAD(R^k_{ik})$ *)
is received on link (i,k)
copy whole routing table into $V^{i,k}$ and send it to k
end

The following function, called IN_PATH, returns "true" if Neighbor is in the path from Node to Dest. Otherwise, the function returns "false."

```
Function IN_PATH(Node,Neighbor,Dest);
(* returns true or false *)
  begin
    h ← HEAD_Node(Dest);
    (* find head from Node to Dest *)
    if h = Node then
    (* Neighbor is not in R_NodeDest *)
      return(false)
    else if h = Neighbor then
    (* Neighbor is in R_NodeDest *)
        return(true)
      else
        IN_PATH(Node,Neighbor,HEAD_Node(h));
        (* cannot determine yet, try again *)
  end;
```

The following procedure, called RT_UPDATE, update the routing table. Any destination j will be assigned a distance copied from column k iff D^k_{ij} is the minimum among row j of distance-table and each node v extracted from D^k_{ij} must be that D^k_{iv} is also the minimum among row v.

Procedure RT_UPDATE;

```
  begin
    initialize all destinations to be unmarked
    for any unmarked destination j do
    begin
      if there is no determined distance in row j
      then mark j as undetermined
      else begin
          TV←ø
          pick up any minimum distance D^b_ij
          c←h^b_ij, TV←TV∪{c}
          repeat c←h^b_ic, TV←TV∪{c}
            until D^b_ic is not minimum of row c or
            h^b_ic=i or h^b_ic is marked
          if h^b_ic is marked as undetermined
              or D^b_ic is not minimum of row c
          then mark each node in TV as undetermined
            else begin
              mark each node in TV as determined
              RDIST_i(j)←D^b_ij; HEAD_i(j)←h^b_ij; P_ij←b
              end
          end
      end
    copy routing table to V^i
  end
```

For the routing table updated by procedure RT_UPDATE, a path which can be extracted from any finite distance in routing table can also be extracted from some column in the distance table at the same node.

4. CORRECTNESS OF ALGORITHM 2

4.1. Bouncing Effect and Counting-to-infinity Behavior

Property 1. At any moment in computation, the path extracted from any distance maintained at each node, say i, to any destination, say j, is a simple path and is equivalent to the path implicit in such distance.

Proof: Let t_0 be the initial moment. Let $\{t_1,...,t_n\}$ be the set of all the moments such that, at each moment, there is at least one node receiving a message, detecting a failure/recovery, or generating a message.

Initially, at time t_0, we require each node, say i, maintain only the distances about each of its neighbors, say j, in its distance table such that $D^j_{ij}=d_{ij}(t_0)$ and $h^j_{ij}=i$ where $d_{ij}(t_0)$ is the initial weight assigned on link (i,j). Therefore, for any distance D^j_{ij} the extracted path is ij which is a simple path and is equivalent to the implicit path of D^j_{ij}. In addition, in each row j, there is only one finite distance D^j_{ij} if j is a neighbor of i. Otherwise, row j has only infinite distances. Therefore, in routing table, $RDIST_i(j)=D^j_{ij}=d_{ij}(t_0)$ and $HEAD_i(j)=h^j_{ij}=i$. Thus, Property 1 holds initially.

Assume that, at any moment t, $t\le t_x$, $x\ge 0$, Property 1 holds.

It is clear that Property 1 still holds at the nodes doing nothing or generating a message at time t_{x+1}. For any other node, say i, which will either detect a failure/recovery, or receive a message.

Upon detecting a failure, due to step (4) of Algorithm 2, the distance table is updated by deleting a column. This will not cause the distance table lose Property 1. Upon detecting a recovery, say (i,j), due to step (5) of Algorithm 2, a new column will be created with a single entry (j,D^j_{ij},i) where D^j_{ij} equal the weight of the recovered link (i,j). This is the same as the initial condition. Thus, Property 1 still holds in the distance table of i in this case. Upon receiving a routing vector V^k from a neighbor, say k, the received routing vector must be generated at some moment t, $t\le t_x$. As assumed, the routing table, from which V^k is generated, has Property 1. Because of function IN_PATH and triple setting up in step (3) of Algorithm 2, any finite distance D_{kj} to a destination in V^k must have extracted the path to be the same as the path extracted from the routing table of node k. In addition, $D_{kj}=RDIST_k(j)$. Note that, since Property 1 holds in the routing table of node k when V^k is generated, the path extracted from $RDIST_k(j)$ is the same as the implicit path of $RDIST_k(j)$. This implies that the path extracted from D_{kj} is the implicit path of D_{kj}. When i processes V^k, due to step (1) of Algorithm 2, the path extracted from D^k_{ij} is the implicit path of D^k_{ij}, because this distance is really passed over link (i,j), and the extracted path is the implicit path of D_{kj} argumented with i. Moreover, because of function IN_PATH, the implicit path of D_{kj} will not have node i in it. Thus, the path extracted from D^k_{ij} after step (1) of Algorithm 2 is simple. Therefore, Property 1 holds in the distance table of any node at any time t, $t\le t_{x+1}$.

Due to procedure RT_UPDATE, as mentioned, the path extracted from any finite distance copied from some column, say m, in the routing table updated is the same as the path extracted from such distance in column m. Because node i has a distance table with Property 1 at all the possible cases as discussed above, i.e., the extracted path equals to the implicit of any finite distance maintained in the distance table, thus, Property 1 also holds in i's routing table. This completes the proof.

Q.E.D.

Theorem 1. Algorithm 2 is without bouncing effect.

Proof: This can be proven directly from Property 1. Because the implicit path of any distance maintained is equivalent to the the path extracted from such a distance, and because the extracted path is guaranteed to be a simple path, the implicit path of each distance maintained is a simple path. Because it is impossible to maintain a distance corresponding to a non-simple path (which is the only case where the bouncing effect occurs), Algorithm 2 is without bouncing effect.

Q.E.D.

As explained earlier, counting-to-infinity behavior is a consequence of bouncing effect. Having no bouncing effect automatically implies having no counting-to-infinity behavior. However, a deeper discussion of how counting-to-infinity is avoided can be made by determining the upper bound of any distance maintained by running Algorithm 2, and how this upper bound is implied. The following proof addresses this issue.

Theorem 2. Algorithm 2 has no counting-to-infinity behavior.

Proof: From the definition made for the implicit path, we know that the corresponding distance must be the summation of the weights of all the links in the path. In addition, due to Property 1, the path extracted from any distance maintained must be a simple path and the same as the implicit path of such distance, any distance maintained

must be bounded by the summation of the top N-1 highest weights experienced. Therefore, none of the distances maintained will have unbounded value. This implies that Algorithm 2 has no counting-to-infinity behavior.

Q.E.D.

4.2. Convergence

In the following, we prove that Algorithm 2 terminates correctly, that is when algorithm terminates, the distance to any reachable node maintained in each routing table is the shortest distance of the final graph and the distance to any unreachable node is marked as infinity.

Theorem 3. Algorithm 2 terminates in finite time after the occurence of last topological change.

Proof: By contradiction. Assume that Algorithm 2 does not terminate. There must be a infinite number of messages sent after the last topological happened. Among these infinite number of messages, there must be infinite messages with finite distances. The reason is that infinite messages with infinite distances only cannot occur because there is a finite number of total distance-table entries of all nodes in the final graph. Moreover, due to Property 1, the path extracted from any distance maintained must be a simple path and must be the same as the implicit path of such distance; hence, the domain of all the possible distances will be included in all the cases in which each case is the total weights of no more than N-1 different links ever experienced. One can see that the number of all such cases is finite. Thus, there must be some distance D_{ij} sent an infinite number of times, because the number of all possible distances is finite. Consequently, there must be a neighbor b that sends i an infinite number of messages that makes i send messages forever.

Each time node i sends D_{ij}, this is caused either when node i receives D_{bj} from b and $D_{ij}=D_{bj}+d_{ib}$ where d_{ib} is weight of link (i,b) at that time, or when D_{ij} has been in node i's distance table of node i at the time it receives a message from b.

If the first case happens an infinite number of times, then node b sends D_{bj} infinite times and $D_{bj}=D_{ij}-d_{ib}<D_{ij}$ because $d_{ib}>0$.

Otherwise, the second case must happen an infinite number of times. In this situation, if D_{ij} is not stable, this means that D_{ij} is changed forever, which is similar to the first case, in which there must be a neighbor b' such that b' sends $D_{b'j}$ an infinite number of times and $D_{b'j}=D_{ij}-d_{ib'}<D_{ij}$, because $d_{ib'}>0$. Else, if D_{ij} becomes stable, then a node i must receive an infinite number of times some distance which is shorter then D_{ij}.

Consequently, there must be a neighbor b" sending $D_{b"j}$ infinite times and $D_{b"j}=D^{b"}_{ij}-d_{ib"}<D_{ij}-d_{ib"}<D_{ij}$ because $d_{ib"}>0$.

Therefore, in any of the possible cases, there must be a node that will infinitely generate messages with a distance at least w less than D_{ij} an infinite number of times, where w is the minimum weight of the final graph. By recursively applying the above argument, this contradicts the fact that all the distances maintained are positive.

Q.E.D.

To prove the correctness of distance-table maintained, we need the following property. The weight of link (x,y), maintained at any node i, which can be derived from i's distance table is $D^k_{iy}-D^k_{ix}$ where $h^k_{iy}=x$.

Property 2. When Algorithm 2 terminates, any link weight maintained in the distance table must be in the final graph.

Proof: We can know from Property 1 that any link weight maintained must be in the history of computation; hence, we only have to consider, in this proof, those link weights having been existent and not in the final graph (i.e., the weight that is changed at some time during the computation). Let d_{uv} be any such weight. We want to show by induction that any node which can reach node u by k hops will not maintain d_{ib} in its distance table. Number k is the index of the induction steps.

It is clear that each node will maintain all the adjacent links correctly. This means that if link (u,v) is not in the final graph, then u must have detected the failure of (u,v) and deleted the column corresponding to neighbor v. In addition, if the final weight d'_{ib} is different from d_{ib}, then u must have been notified about the weight change; therefore, it must have kept the weight of link (u,v) to be d'_{ib}.

Assume that all the nodes k hops away from u, $k\geq 0$, will not maintain d_{ib} in their distance tables.

Let's check any of the nodes k+1 hops away from u. Since all the nodes no greater than k hops away from u will not maintain d_{ib} in their distance table, none of these nodes will maintain d_{ib} in their routing tables (the link weights derived from the routing table is a subset of the ones derived from distance table). Therefore, if a node i k+1 hops away from u maintains w(u,v) in its distance table, there must be a neighbor of i which is no less then k+1 hops away from u

and maintains d_{ib} in its routing table. Without loss of generality, let i_1 be the neighbor which offers i the minimum distance to go to v among the neighbors offering i with d_{ib} and i_x be the preferred neighbor of i_{x-1} where $x \geq 2$. If there is a node i_y, $1 \leq y < x$, which is a preferred neighbor of x, then we will have $D_{yv} < D_{y+1v} \ldots < D_{xv} < D_{yv}$ which is impossible. Therefore, the preferred neighbors chain of node i_1 must be a list of distinct nodes. However, we have only a finite number of nodes no less than k+1 hops away from u and each i_x must have a preferred neighbor to offer d_{ib}. Because this is impossible, any node k+1 hops away cannot maintain d_{ib} in its distance table, either.

Q.E.D.

Theorem 4. When Algorithm 2 terminates, the distance for any node i to any other reachable node j maintained in the routing table of i is the shortest distance between i and j in the final graph and the preferred neighbor will be maintained correctly; also, the distance from i to any unreachable node is marked as undetermined.

Proof: Due to Property 1, for any finite distance maintained at any node i to any other node j, we can always extract a complete path from i to j. Therefore, if a finite distance to an unreachable node is maintainedand, there must be a link in this path whose weight is not in the final graph. This violates Property 2; therefore, there can not be any distance about j maintained in the routing table of node i.

For an arbitrary node j reachable from an arbitrary node i, there will be shortest paths from i to j. Note that these paths can have different number of hops and let n_j is the maximum among these. Also, for convenience, let d_{uv} be the weight of the link from u to v in the final graph.

Again, we will prove by induction indexed by n_j.

If n_j is 1, then j is a neighbor of i and the shortest path is unique with distance d_{ij}. Since j is a neighbor, i will maintain $D_{ij}^j = w(i,j)$ and $h_{ij}^j = i$ in column j. Moreover, if there is any other distance maintained in row j, then Properties 1 and 2 ensure that such a distance must be for a existent path due to Property 1 and Property 2. Since we know that the shortest path is unique, d_{ij} must be the only minimum in row j. Therefore, it will be recorded in the routing table of i because of FunctionRT_UPDATE with $P_{ij}=j$.

Assume that when $n_j \leq k$, $k \geq 1$, i will maintain the correct distance and preferred neighbor in its routing table.

If n_j is k+1, there must be a neighbor i' of i such that $n_{j'}=k$ and $\underline{D}_{ij}=\underline{D}_{i'j}+d_{ii'}$ where $n_{j'}$ is defined in the same way for i' than n_j is for i , and $\underline{D}_{ij}$ and $\underline{D}_{i'j}$ are the true shortest distances for i and i' to go to j, respectively. Note that, because $\underline{D}_{ij}=\underline{D}_{i'j}+d_{ii'}$, i' can not have a shortest path to go to j through i. Moreover, $RDIST_{i'}(j)=\underline{D}_{i'j}$, because $n_{j'}=k$. Therefore, $D_{ij}^{i'}=RDIST_{i'}(j)+d_{ii'}=\underline{D}_{ij}$ which is the minimum in row j. For any minimum D_{ij}^b in row j (b can be i'), the path it represents is always a shortest path, because each distance maintained in any distance represents an existent path. In addition, any node x in the shortest path P from i to j must also have the subpath from x to j on P as its shortest path. Therefore, the last operation for i to update $RDIST_i(j)$ using Function RT_UPDATE, and arbitrarily picking up a minimum will always have a successful trace for j and thus have $RDIST_i(j)=D_{ij}^{i'}=RDIST_{i'}(j)+d_{ii'}=\underline{D}_{ij}$ and $P_{ij}=b$.

Q.E.D.

5. LOOP FREEDOM

Algorithm 2 as described in Section 3 cannot guarantee loop freedom (see Section 1) at every instant. This can be illustrated by the following example, illustrated in Fig.3.

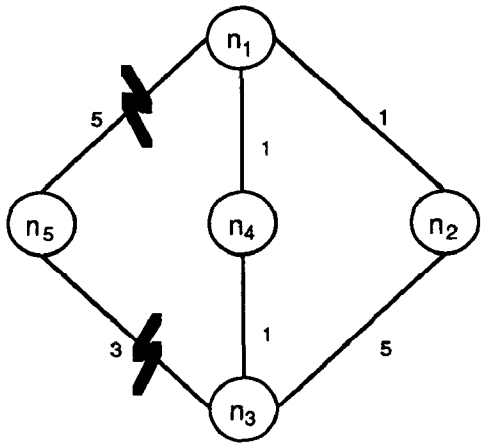

FIG 3 -- A network where links (n_3,n_5) and (n_1,n_5) failed about the same time.

In Fig. 3, consider node n_5 as the destination. Assume that nodes n_1 and n_3 detect the failures of (n_3,n_5) and (n_1,n_5) at about the same time. Based on Algorithm 2, nodes n_1 and n_3 will choose node n_2 and n_4 as the new preferred neighbors, respectively. Thus, a loop, (n_1,n_4,n_3,n_2,n_1), is formed. However, we know that nodes n_1 and n_3 will send routing vectors to both neighbors n_4 and n_2. Assume that the message delay for counter-clockwise direction of the loop is very short whereas it is very long in the clockwise direction. Upon receiving the routing vector through counter-clockwise direction, nodes n_4 and n_2 will not change their preferred neighbors. If there is any data packet sent counter-

clockwise, it can loop many times until the routing vector from clockwise direction arrives to nodes n_4 and n_2. At this moment, nodes n_4 and n_2 can find out that node n_5 is not reachable and then break the loop.

In fact, a loop can not be triggered by only recoveries in either Algorithm 1 or 2 proved by Jaffe and Moss [13] for the case of Algorithm 1. Therefore, if a loop, say n_0, n_1, ...,n_x, n_0, is formed at time any t, then there must be at least one node, say n_0, which has lost its current preferred neighbor upon receiving routing vector, triggered by a failure notification. (In our case, only multiple failures can cause loops.) Then, n_0 switches to a new preferred neighbor to form the loop. Hence, a routing vector must be sent by n_0 to all the neighbors. As can be seen from the above example, the loop will be broken before the routing vector, issued by n_0, traverses back to n_0. Since we know that the routing vector will not keep on going on a loop (because of Property 1), it is enough for us to design a scheme to prevent routing-table loops. Thus, we only have to set up an independent rule to control the sending of data packets without changing the basic protocol (Algorithm 2). Routing-table loops can be prevented in the following way. When a node sends routing vectors to its neighbors , it waits for the response from all its neighbors, then forwards the data packet to the preferred neighbor. In other words, during the waiting time for the response, the data packet will be held. To implement this rule, we have to let each neighbor respond with an acknowledgement even if its routing table is not changed upon receiving the routing vector. However, this does not affect our basic protocol. This interneighbor coordination approach is also adopted by Garcia-Luna-Aceves [10] to prevent routing-table loops in the context of a minimum hop routing algorithm . The formal proof for the prevention of routing-table loops is in [10].

6. PERFORMANCE ANALYSIS

The number of messages generated by Algorithm 1 is bounded by an exponential function of N, a polynomial function of the degree of G(V,L), and a linear function of the number of topological changes [21]. All protocols based on the BF algorithm, including this paper, can not get rid of this drawback. However, the advantage of these protocols lies in their low time complexity. The time complexity for single failure/recovery has been reduced from O(N) (assuming that no bouncing effect is encountered) of the original BF algorithm to O(h) for the case of single resource failure where h is the height of the shortest-path tree, by Jaffe, Moss and Garcia-Luna-Aceves [9,10,13]. The other approaches without using the BF algorithm always result with time complexity $O(h^2)$ [7,16].

The time complexity for Algorithm 2 is O(h) for single link failure/recovery, and is O(N) for multiple failures and recoveries the same as in [9,10,13]. However, this implementation is without the overhead of maintaining the bit vector [13] in each node for each neighbor for the updating processes, caused by the multiple changes, in which these nodes are involved. In the case when network becomes disconnected, our algorithm can detect disconnectivity much faster than the other approaches. (For instance, the disconnectivity caused by single failure can be detected right away.) Moreover, the storage required for each node is the same as BF algorithm except that one extra field, the head of the path, in each entry of the distance table and the routing table is needed.

It is clear that, if no bouncing effect or counting-to-infinity problem is encountered, the number of steps needed is O(N) for Algorithm 1. Algorithm 2 also has O(N) time complexity without any further enhancement. This is illustrated through Fig. 4.

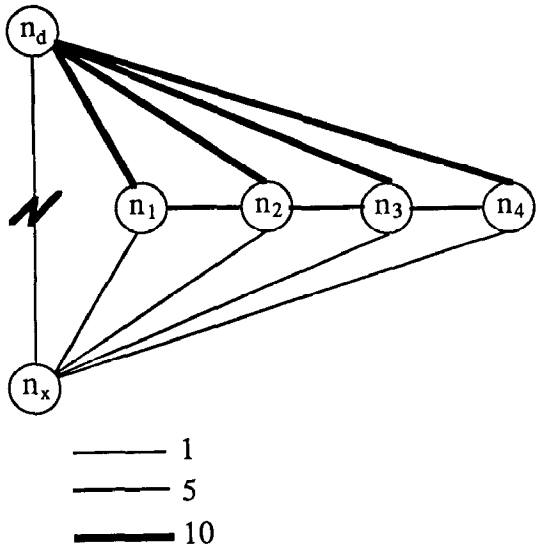

FIG 4 -- A network where link (n_x,n_d) failed

In Fig. 4, assuming that n_d is the destination, nodes n_1, n_2, n_3, and n_4 will all have shortest paths going through n_x to n_d before link (n_x,n_d) fails. When link (n_x,n_d) fails, nodes n_1, n_2, and n_3 adapt their shortest paths to go through nodes n_2, n_3, and n_4, respectively. This is caused by the fact that each of them does not know that the new selected path is also broken by the same failure. Thus, node n_1 will adapt to link (n_1,n_d) only after receiving a routing vector from n_2 (where n_4 first sent a vector to n_3, and then, n_3 sent one to

n_2). Obviously, if we let network G(V,L) have the same topology as Fig. 4, G(V,L) has the N-2 nodes located between n_x and n_d, then the number of steps needed for node n_1 to choose the proper neighbor is O(N).

However, the time complexity of the presented protocol can be improved to achieve O(h), where h is the maximum height of the trees experienced during the computation, by adding only one more rule. Observing that the critical condition to cause O(N) is that a node may switch to a path in which the failed link is also located, if we could stamp all the messages triggered by a topological change (n,m) with (n,m) as the event identity, then, at the moment a node loses its current preferred neighbor, it will switch to the neighbor that offers the shortest distance representing a path which does not contain the failed link. Recall that each node can determine the entire path corresponding to each entry in the distance table. This additional rule, which checks if the link identified as the triggering event is on the path, is easy to implement without requiring any extra local information. This is formally argued in Theorem 5. We will refer to the refined protocol, which is the basic protocol mentioned in Section 3, along with the interneighbor coordination mentioned Section 5, and the event identity mentioned above as the *enhanced* protocol.

Theorem 5. The time complexity for a single link failure for the *enhanced* protocol is O(h) where h is the maximum height experienced during the computation.

Proof: Fix the destination to be n_d. Let the failed link be (n,m) and node m is downstream (in sink tree, i.e., closer to the destination) to node n.

The node with initial shortest path without going through the failed link will not change its routing table since the original shortest is not changed and the failure can only increase the distance for other alternative paths. Let's call this kind of node to be a stable node.

The node n_i, with original shortest path going through the failed link (n,m) and i hops away from node n on the initial shortest path, will receive a routing vector with messages stamped with event identity (n,m) from the original preferred neighbor. It is because that node n will change the routing table by selecting the new preferred neighbor after it lost the original preferred neighbor m by detecting the failure of its adjacent link toward node m. Then, since the routing table of node n is changed, the routing vector generated by failure will be sent to all the neighbors. The neighbor which will change the routing table or, equivalently, has the initial shortest path involving (n,m) will also send subsequent routing vectors to all the neighbors. Based on the fact that if n_i is going to change its routing table then so are all the nodes lie between n and n_i. Thus, actually, n_i will receive a routing vector from its original preferred neighbor in i steps. Furthermore, there must be at least one node, in the final shortest path tree rooted by n_d, selecting a *stable* node as the new preferred neighbor, otherwise even a spanning tree will not be constructed. We will call this kind node to be *boundary* node. Based on the above description, each boundary node acting as n_i will receive a routing vector with stamped messages in i steps. Then, it follows that each boundary node select that *stable* node immediately because it can determine the alternatives passing through (n,m) by checking its distance table. Similarly, the nodes, which should be upstream and j hops away from it in the final shortest path spanning tree, will receive the correct distance in j steps.

Notice that both i and j mentioned above are bounded by the height of initial tree and the final tree. Thus, the steps for a node to converge to the correct distance is O(h).

Q.E.D.

Another factor determining the performance of a distributed routing algorithm is the nodal computation time which becomes important in very high-speed network and very large network. Here, we mainly discuss the time needed to update the routing table by comparing the procedure RT_UPDATE and the breadth first search approach suggested by Humblet [12]. As can be seen, by doing breadth first search, each entry of distance-table has to be considered when the shortest path to a destination is the longest among the paths to all the leaves in the tree. The minimum time needed to process each entry is logN, because one has to determine if such destination has been reached, and if not, insert the distance into the ordered list maintained for breadth first search. However, by running procedure RT_UPDATE, even without sophisticated data structure, each entry will be processed eaxctly once using one unit computation time (compared to logN units).

In addition, due to the function IN_PATH, the destinations located behind the neighbor in the sender's shortest path tree will not be sent to that neighbor. Therefore, the size of the routing vector and the nodal computation storage can be smaller.

7. CONCLUSION

In this paper, we present a protocol that avoids the

undesirable effects of bouncing, counting-to-infinity behavior, and routing-table looping of the distributed Bellman-Ford shortest path algorithm. The number of messages needed is no more than that of the distributed BF algorithm (possible to be exponential [21]). The time complexity is O(h) for single link failure or recovery, the same as in [9,10,13], and O(N) for multiple failures and recoveries.

An open question is that if a O(h) time complexity algorithm is existent. In addition, whether O(N) time complexity and polynomial message complexity can both be achieved.

REFERENCES

[1] B. Awerbuch and S. Even, " Reliable Broadcast Protocols in Unreliable Networks," *Networks*, vol. 16, no. 4, pp. 381-396, Dec. 1986.

[2] D. Bertsekas and R. Gallager, *DATA NETWORKS*, pp. 297-333, Prentice Hall, Inc., 1987.

[3] C. Cheng, I.A. Cimet, and Srikanta P.R. Kumar, "A Protocol to Maintain a Minimum Spanning Tree in a Dynamic Topology," *ACM SIGCOMM Symp. Commun. Arch. and Protocols*, pp. 330-338, Stanford, CA, Aug. 1988. Also in *Computer Communications Review*, Vol 18, no. 4.

[4] I.A. Cimet, C. Cheng, and Srikanta P.R. Kumar, "On the Design of Resilient Protocols for Spanning Tree Problems," to appear in *IEEE Int'l Conf. on Distributed Computing*, June 1989.

[5] K.C. Chu, "A Distributed Protocol for Updating Network Topology Information," Research Report, RC 7235, IBM Thomas J. Watson Center, July 7, 1978.

[6] Narsingh Deo, *Graph Theory with Application to Engineering and Computer Science*, pp.20, Prentice Hall, Inc..

[7] E.W. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numer. Math.*, vol. 1, pp. 269-271, 1959.

[8] J.J. Garcia-Luna-Aceves, "A Fail-Safe Routing Algorithm for Multihop Packet-Radio Networks", *IIEEE NFORCOM '86 Proceedings* , Miami FL., Apr. 1986.

[9] J.J. Garcia-Luna-Aceves, "A Unified Approach to Loop-Free Routing Algorithm Using Distance Vectors or Link States," *ACM SIGCOMM Symp. Commun. Arch. and Protocols*, Austin, Texas, Sep. 1989.

[10] J.J. Garcia-Luna-Aceves, "A Minimum-Hop Routing Algorithm Based on Distributed Information," *Computer Networks and ISDN Systems*, vol 16, pp. 367-382, May 1989.

[11] Jacob Hagouel, "Issues in Routing for Large and Dynamic Networks", *IBM Research Report* RC 9942 (No. 44055) Communications, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, Apr. 1983.

[12] Pierre A. Humblet, "An Adaptive Distributed Dijkstra Shortest Path Algorithm", an unpublished paper, Feb. 1989.

[13] J.M. Jaffe and F.M. Moss, "A Responsive Routing Algorithm for Computer Networks", *IEEE Trans. Comm.*, vol. COM-30, no.7, pp. 1758-1762, July 1982.

[14] J. McQuillan and D.C Walden, "The ARPANET Design Decisions," *Computer Networks*, vol. 1, Aug. 1977.

[15] J. McQuillan, I. Richer, and E.C. Rosen, "The New Routing Algorithm for the ARPANET," *IEEE Trans. Comm.*, vol. COM-28, May 1980.

[16] P.M. Merlin and A. Segall, "A Failsafe Distributed Routing Protocol," *IEEE Trans. Comm.*, vol. COM-27, pp. 1280-1288, no. 9, Sep. 1979.

[17] R. Perlman, "An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN," *9th Data Comm. Symp.*, Aug. 1985.

[18] R.D. Schlichting and F.B. Schneider, "Fail-Stop Process: An Approach to Designing Fault-tolerant Computing Systems," *ACM Trans. Comput.*, pp. 222-238, Aug. 1983.

[19] K.G. Shin and M. Chen, "Performance Analysis of Distributed Routing Strategies Free of Ping-Pong-Type Looping," *IEEE Trans. Computers*, vol. COMP-36, no. 2, pp. 129-137, Feb. 1987.

[20] M.S. Sloman and X. Andriopoulos, "A Routing Algorithm for Interconnected Local Area Networks", *Computer Networks and ISDN Systems*, pp. 109-130, 1985.

[21] W.D. Tajibnapis, "A Correctness Proof of a Topology Information Maintenance Protocol for a Distributed Computer Network," *Comm. of ACM*, vol. 20, pp. 477-485, 1977.