Algorithms for Finding Disjoint Paths in Mobile Networks

Sumit Arora *Hyunyoung Lee* *Ramakrishna Thurimella**

Department of Computer Science
University of Denver, Denver CO 80208

Abstract

Disjoint paths are useful in mobile networks for fault tolerance, increasing bandwidth, and achieving better load balance. Communication over multiple disjoint paths is a less expensive alternative to flooding the network. We present a distributed algorithm for finding k disjoint paths, for any given k, between a source S and a destination T in mobile networks where the links have uniform weight. Our algorithm runs in $O(kn)$ time using $O(km)$ messages where n is the number of nodes and m is the number of links in the network, and k is the number of paths to be found. We also present a resilient version of this algorithm that may be suitable for networks whose topology changes frequently.

1 Introduction

In mobile ad hoc networks (MANETs), nodes assume the role of a router as there is no dedicated routing infrastructure. Due to mobility, however, the network connectivity changes frequently. By maintaining multiple disjoint paths, one can increase the probability that source can reach the destination *via* one of the known paths as the network undergoes topological changes. Multiple paths can also be employed to achieve better load balance and improve quality of service in bandwidth-constrained MANETs. For fault tolerance and energy conservation, communication over multiple disjoint paths is clearly a less expensive alternative to flooding the network.

For these and other important reasons, the problem of finding disjoint paths in communication networks attracted a lot of attention. The obvious approach to pursue is to generalize the known routing algorithms to multiple disjoint paths. The distance vector protocols, based on Bellman-Ford algorithm, are distributed implementations of all-pairs shortest path algorithms. For details, refer to [H00]. Protocols such as Routing Information Protocol (RIP) [H88] use the property that for every pair of communicating hosts S and T, and intermediate node V, the shortest path between S and T contains shortest paths between S and V, and V and T. This key property lends these algorithms for simple and elegant distributed implementations. No such property exists for disjoint paths and hence it is not possible to generalize Bellman-Ford type algorithms to multiple disjoint paths in a straightforward manner.

In link state algorithms [C89], on the other hand, nodes collect snapshots of the state of the network and run sequential algorithms to determine the best routes to use for communication. Such an approach can be used in principle to find multiple disjoint paths, but the overhead in collecting snapshots can be prohibitive in a network that is either large or under constant change.

In this paper, we consider the problem of finding k node-disjoint paths, for any given k, between a source S and a destination T in mobile networks where the links have uniform weight. Sequential algorithms for k disjoint path problem have been known for many years. At a high level, the algorithms for the node disjoint case have these three key steps in common:

*This research was supported by NSF grant CCR-9821022.

1. a *bidirectional* network is constructed by replacing each undirected link by two oppositely oriented directed links,

2. the *split* graph is obtained from the original by replacing each node V with two subnodes V_i and V_o, adding the edge $V_i \rightarrow V_o$, connecting all incoming (resp. outgoing) links to V_i (resp. V_o), and deleting S_i and T_o,

3. a weight of 1 is associated on each link and the maximum flow between S_o and T_i is found in the resulting 0/1 directed network (denoted G_d).

This method (referred henceforth as the Augmenting Path Algorithm) can be generalized to networks with nonuniform link delays by attaching costs to links and running minimum-cost-maximum-flow algorithm in Step 3. While minimum-cost-maximum-flow algorithms can be complicated for practical implementation, finding maximum flow in 0/1 networks is rather simple and can be implemented by an iterative augmenting path approach. An augmenting path can be found by performing a breadth-first or depth-first search in what is known as the residual graph.

Even though the methods sketched above have been known for many years, they never found their way to the domain of network protocols. We speculate the reason for this gap between theory and practice to be the perceived difficulty of graph transformation involved in the known sequential algorithms. We attempt to bridge this gap by providing simple, distributed implementation of the Augmenting Path Algorithm. The data structures used by our algorithms are very elementary and suitable for practical implementation. Our algorithm runs in $O(kn)$ time where n is the number of nodes in the network and k is the number of paths to be found. The message complexity is $O(km)$ where m is the number of links in the network. We also present a resilient version of this algorithm that can compute disjoint paths as the underlying topology is changing.

This paper is organized as follows. After reviewing related work in the next section, we discuss problem formulation, our assumptions, and a description of Augmenting Path Algorithm in Section 3. Section 4 presents our distributed implementation. The resilient version is given in Section 5. We conclude the paper in Section 6.

2 Related Work

There has been considerable recent interest in the mobile computing community in the disjoint path problem. Multipath and alternate path routing performance was the subject of investigation in [NCD01, PHST00, VSR03, WH01]. Papadimitratos et al. introduce a heuristic to find k disjoint paths that are reliable [PHS02]. Ye et al. propose a variation of Ad Hoc Distance Vector (AODV) routing protocol that can withstand node failures. Disjoint routing was studied in [LG01, VG01].

Itai and Rodeh [IR84] introduced the notion of k-independent trees–k spanning trees rooted at node r such that the k tree paths from any other node v to r are disjoint–and gave a distributed algorithm to find such trees for $k = 2$ in 2-edge connected graphs. For $k > 2$, they further conjectured that there exist k independent trees if and only if the underlying network is k connected. For general k, this conjecture is still open. Ever since, several empirical algorithms have been proposed to find k independent spanning trees. One such result is by Sidhu et al. [SNA91]. They give distributed algorithms that is a generalization of this approach. They show how to find multiple trees to a given destination r with a guarantee that one of the trees is a shortest-path tree. Furthermore, for every v, paths between v and r are disjoint. The addition of the requirement that one of the trees be a shortest path tree changes the problem as the example in Figure 1 shows: There are two disjoint paths between S and T: $SABCT$ and $SDEFT$, each of length 4. However,

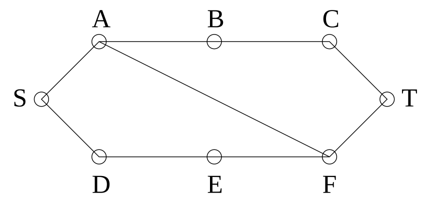

Figure 1: Requiring one of the paths be a shortest path limits the number of disjoint paths

with the additional requirement that one of the paths be shortest, there is only one path $SAFT$, of length 3.

Chen et al. [CDS98] proposed a solution to the same problem, but without requiring that one of the paths be a shortest path. Their algorithm finds multiple paths, if the flooded messages that are used for finding paths arrive by disjoint paths. In other words, there are networks in which their algorithm finds multiple paths, but not *all* available disjoint paths between a pair.

Cheng et al. [CKG90] suggested a distributed algorithm for finding k disjoint paths of least weight between a pair of nodes in a given weighted network. Though more general than the problem considered in this paper, finding shortest paths introduces the complications associated with detecting termination and convergence. As a result, their algorithm is considerably more complex than ours and may be unsuitable for implementation in MANETs.

Other relevant work [ORS93] includes algorithms for finding two disjoint paths of minimum total cost from every node to a destination. Our work differs from theirs in two respects: first, our algorithm works for any value of k; second, the graph transformation that is necessary here is significantly simpler than the one presented in their paper.

3 Preliminaries

3.1 Notation and Assumptions

We model the mobile communication network as an undirected, unweighted graph. We assume that there is at most one link between any pair of nodes, and each node is aware of only its neighbors. We describe computing disjoint paths for one pair of nodes S and T, with the understanding that several such computations could be taking place simultaneously.

For a (sub)graph G, $E(G)$ denotes the set of edges in G. For two sets A and B, the symmetric difference $A \oplus B = (A - B) \cup (B - A)$.

In a directed graph G with source S and destination T and a given set of disjoint paths $\mathcal{P}$ from S to T, the *residual* graph G_r is obtained by reversing the edges of $\mathcal{P}$ in G. *Augmenting path* in G_r is any directed path from S to T. For these and other concepts related to network flows, the reader is referred to [CLRS01].

In our description of algorithms for the node disjoint case, each node U, except for the source S and the destination T can be conceptually viewed as a pair of neighboring subnodes - *in* subnode U_i and *out* subnode U_o with a directed edge going initially from U_i to U_o. Each node runs two instances of the algorithm, one for each subnode. The source S (resp. destination T) runs only the instance corresponding to the *out* (resp. *in*) subnode. In our notation, uppercase letter I represents a node, whereas lowercase letter i represents either *out* subnode I_o or *in* subnode I_i of I.

Flooding refers to each node sending a message to all its neighbors in the directed networks that our algorithms construct during execution. On the other hand, we use the term *broadcast* in the same way it is used in conventional networking literature, i.e. the operation is performed on the original (undirected) communication network.

Algorithm AugmentingPath
Input: A graph G, source S, destination T, and the desired number k of disjoint paths
Output: Available number of disjoint paths, up to k, between S and T

1 build the directed graph G_d corresponding to G as described in Section 1
2 $G_r := G_d$ // initialize the residual graph G_r
3 i := 0; path_exists := true; $\mathcal{P} := \emptyset$
4 while (path_exists & i < k) {
5 attempt to find a directed path P_d between S and T in G_r
6 if no such path exists assign path_exists := false
7 else reverse the edges of P_d in G_r // update the residual graph
8 let P be the path in G that corresponds P_d
9 increase the number of disjoint paths by doing $E(\mathcal{P}) := E(\mathcal{P}) \oplus E(P)$
10 i := i+1
11 }

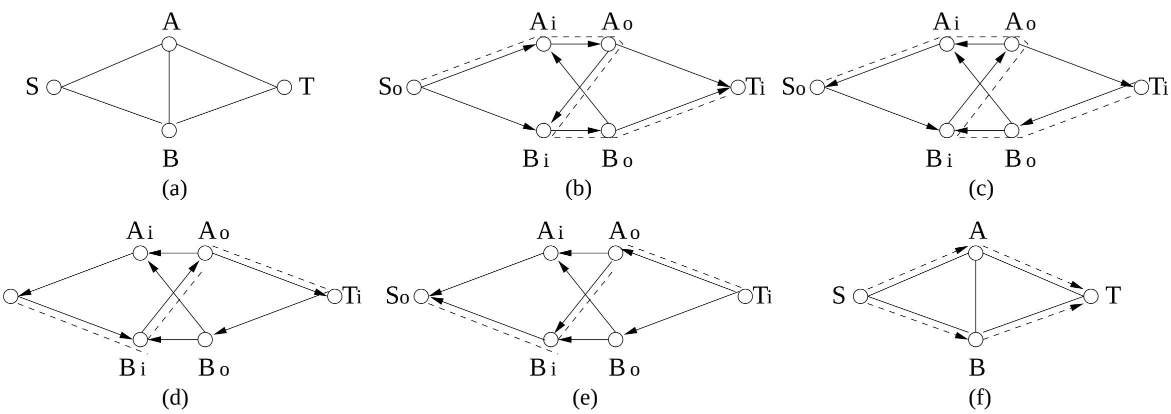

Figure 2: AugmentingPath algorithm and its execution

3.2 The Sequential Algorithm

In this section we present AugmentingPath, a sequential algorithm based on augmenting paths, and illustrate an example execution of it. The distributed algorithm which we present in the next section simulates the sequential algorithm by use of some auxiliary data structures.

Figure 2 presents the pseudocode of AugmentingPath algorithm and an execution of it. The graph in (a) is the original graph. The split graph G_d is shown in (b). G_r is initialized to G_d in line 2 of the algorithm. The dotted line in (b) represents the first directed path ($S_oA_iA_oB_iB_oT_i$) found by line 5 of the algorithm. By reversing the path (line 7) we get the updated G_r in (c). In the next iteration of while loop, the second path ($S_oB_iA_oT_i$) is found, which is represented as the dotted line in (d). In (e), the second path is reversed, and now we can see that there are no more outgoing edges from the source node, which means that there are no more paths to be found. Finally we have found two disjoint paths: SAT and SBT as shown in (f).

4 The Algorithm

In this section we present a distributed algorithm to find k disjoint paths between a pair of nodes, which is a distributed version of the AugmentingPath Algorithm. For now, we assume the topology to be static and the network to be reliable. We relax this assumption in the following section, where we present the resilient version of the algorithm presented in this section. We refer to the algorithm in this section as the basic algorithm.

The goal of our algorithm is, given a pair of nodes, to find as many as possible, up to k, disjoint paths, for any given k. The algorithm works as follows. The source node S initiates the path discovery by sending the path discovery token (PDT) message to its neighbor(s). When a node, other than the destination, receives a PDT message, it starts participating in the path discovery, by creating the data structures needed in the algorithm (described below). Then the intermediate node forwards the PDT message to its neighbor(s). Forwarding the PDT to the neighbor(s) is accomplished in the function Explore(). We present two implementations of Explore(): Explore-DepthFirst() and ExploreBreadthFirst(). The former is easier conceptually, while the latter is more time efficient.

The Explore() function searches the graph much the same way a graph would be explored while building a distributed depth-first or breadth-first spanning tree with source as the root. The difference here is that once the destination node is found (i.e. when a PDT reaches the destination), the search is concluded, and the destination sends a path marker token (PMT) to mark the tree path from the destination to the source. When an intermediate node receives the PMT, it makes appropriate changes in its data structures to reflect the link reversals on this tree path. When the PMT reaches the source, the source starts next iteration by sending out another PDT. The path exploration proceeds in rounds and continues until all k (or all existing) paths are found. Finally, the source sends out the DONE message along the paths, letting the intermediate nodes record the next hop information in their routing table entry for the source and destination pair. The pseudocode of the algorithm is presented below.

To summarize, the algorithm uses three types of messages as follows:

- **PDT** (Path Discovery Token): When a node (source) S wants to find a path to another node (destination) T, it starts the path discovery by sending out a PDT to its neighbor(s) which in turn forward(s) it to their neighbor(s). A PDT contains the following information: source, destination, backtrack flag. If a node cannot find a path to T, it returns PDT with backtrack flag set to true. We denote a PDT with backtrack flag set to true as $\mathrm{PDT_B}$.

- **PMT** (Path Marker Token): When the destination receives a PDT, it sends out a PMT along the newly discovered path. A PMT contains $\langle S, T\rangle$

- **DONE**: After finding the last path, the source sends out a DONE message on all paths with the information of $\langle S, T\rangle$. Every node that receives DONE message makes a routing table entry indicating the next hop for a path from S to T.

Every node (except destination) starts the algorithm by creating the list L of it neighbors. In list L, the entry for neighbor v, L[v] has a direction field. The direction field can have a value either *in* or *out*. We denote the direction of neighbor v in L as L[v].direction, where the direction *in* (resp. *out*) means the directed edge from (resp. to) v is incoming (resp. outgoing) in the residual graph.

For the source node S_o, initially every neighbor's direction is *out* (note that after splitting, the subnode S_i is deleted). Analogously, for the destination node T_i, initially every neighbor's direction is *in*. In the list L of an intermediate node V_i (resp. V_o), the direction of all entries is initially *in* (resp. *out*), except the one for V_o (resp. V_i).

Function ExploreDepthFirst(L)
Input: A list of neighbors in L
Output: If a path can be found to T, return the successor on that path; otherwise 0

```
initially all neighbors are unvisited
while (there is an unvisited neighbor q) {
    send PDT to q
    wait for either PDT_B or PMT from q
        meanwhile if any other message arrives return it to the sender with backtrack flag set
    if the message from q is PMT return q
    else mark q as visited
}
return 0
```

Function ExploreBreadthFirst(L)
Input: A list of neighbors in L
Output: If a path can be found to T, return the successor on that path; otherwise 0

```
for each neighbor q, send a copy of PDT to q
for each neighbor q, wait for either PDT_B or PMT from q
    meanwhile if any other message arrives return it to the sender with backtrack flag set
if one of the neighbors, q, sent PMT return q
else return 0
```

Algorithm for the source node S
Input: A list of neighbors in L and the desired number k of disjoint paths
Output: Up to k routing table entries corresponding k disjoint paths from S to T

```
r := 1; path_exists := true
while (path_exists & r ≤ k) {
    h := Explore (L)
    if (h = 0) path_exists := false
    else L[h].direction := in //reverse edge (S,h)
    r := r+1
}
for each neighbor h such that L[h].direction = in
    add H to the list of next hops in the routing table for ⟨S,T⟩ entry
    send DONE to h
```

Algorithm for the destination node T
Input: A list of neighbors in L

```
when a PDT is received from p
    L[p].direction := out //reverse edge (p,T)
    send PMT to p
```

Algorithm for an intermediate node i
Input: A list of neighbors in L
Output: Next hop entry in the routing table for $\langle S, T\rangle$

```
when a PDT is received from p
    h := Explore(L)
    if (h = 0) send PDT_B to p
    else L[h].direction := in //reverse edge (i,h)
         L[p].direction := out //reverse edge (p,i)
         send PMT to p
when DONE is received from p
    if i is a in subnode of node I, pass it to the corresponding out subnode
    else find the unique h such that L[h].direction = in
         put H as the next hop in I's routing table for ⟨S,T⟩ entry
         pass DONE to h
```

The correctness of the basic algorithm and its complexity can be proved by the correctness and the complexity of the sequential algorithm [CLRS01]. The details are deferred to the full paper.

Theorem 1 *Given source s and destination t, if there exist k disjoint paths between s and t, the basic algorithm finds k such paths.*

Given that we explore the network k times, each time using a breadth-first or a depth-first search, the theorem below follows. Notice that even though the complexity of a breadth-first search is $O(D)$ time where D is the diameter of the network, the diameter of the residual graphs could increase to n.

Theorem 2 *Given source s and destination t, the basic algorithm finds k disjoint paths between s and t in $O(kn)$ time using $O(km)$ messages where n is the number of nodes and m is the number of links in the network.*

Quality of Discovered Paths Obviously, it is preferable to find k disjoint paths in weighted networks where the weights represent latencies or some sort of reliability measure. However, this complicates the distributed algorithms significantly to a point where they are not practical. For this reason, we deliberately chose a network in which the link weights are uniform. But, using breadth-first search and choosing the first path discovered between a source and destination as we have done in our algorithm, we conjecture that the quality of the paths would be competitive to the ones that take the edge weights into account.

5 Disjoint-Path Computation Under Topological Changes

The basic algorithm in the previous section assumed that the network topology was static while the k disjoint paths were discovered. In this section, we generalize this approach to a network that is undergoing constant topological changes. For instance, a mobile node can simply walk away from the communication network. We model such changes by link failures. Note that node failures can be modeled as a special case of a certain set of link failures. That is, if node v were to fail, this event can be modeled as the failure of all links adjacent to v.

When new nodes come in to the network, they are added to the list L of their neighbors and hence they automatically start participating in the path discovery process. The path discovery proceeds in rounds. In the absence of link failures on discovered paths, there would be k rounds and one new path would be discovered in each round. But if there are link failures on discovered paths causing $f \leq k$ paths to be lost, there would be f rounds to erase those failed paths and f additional rounds to find paths to make up for the lost ones, for a total of $k + 2f$ rounds. Each of these $k + 2f$ rounds has a unique sequence number. Any PDT, PDT_{Ack}, PMT or PATH_ERASE message sent in round j has sequence number j. Since all these messages are either broadcast or flooded, a node responds only to the first message it receives (of each kind) with any given sequence number.

The algorithm uses five types of messages as follows:

- **LINK_FAIL**: Whenever a node detects one of its links to be inactive, it broadcasts a LINK_FAIL message containing the corresponding node information. Source node S has an array LF to store LINK_FAIL broadcasts; no other node stores LINK_FAIL broadcasts.
- **PATH_ERASE**: contains the path to be erased and the sequence number.

- **PDT**: contains the path traversed by this PDT, destination T, and the sequence number. Source S begins each round, either by flooding a PDT to all neighbors in residual graph G_r or by broadcasting a PATH_ERASE message, if it has any unprocessed LINK_FAIL messages resulting in path failures.

- $\mathbf{PDT_{Ack}}$: contains the path traversed by the corresponding PDT and the sequence number. Upon receiving a PDT, T_i broadcasts a PDT_{Ack} with the same sequence number as the PDT. Upon receiving a PDT_{Ack}, S_o broadcasts a PMT with the same sequence number as the PDT_{Ack}.

- **PMT**: contains the path traversed by the corresponding PDT and the sequence number. After sending the PMT or PATH_ERASE broadcast, S_o computes the new set of paths, updates the variables *seq* and *path*, updates its routing table entries to reflect the newly discovered path and reverses the direction, in G_r, of edges which are on the new path. After a delay, that is sufficient for the PMT broadcast to reach all nodes, S proceeds to a new round.

Every node maintains the sequence number of the last PMT or PATH_ERASE broadcast it has seen in an integer variable *seq* and the set of all discovered paths in variable *path*. It also maintains the sequence number of the last PDT it has seen in an integer variable seq_{PDT}. If any node fails and later comes back up, it requests its neighbors to send *path* and *seq*; it initializes seq_{PDT} to 0.

An intermediate node responds to a PDT only if it has received all PMT and PATH_ERASE broadcasts of previous rounds. In other words, if a node has missed out on any broadcast messages, it does not participate in path discovery until it is up-to-date with all the broadcasts from S. Upon receiving a PDT, it updates its seq_{PDT} variable, appends itself to PDT.*path* and floods a copy of the modified PDT to all neighbors in G_r.

Upon receiving a PMT or a PATH_ERASE message, if an intermediate node has not received all PMT and PATH_ERASE broadcasts of previous rounds, it stores this message for future use. When it has received all PMT and PATH_ERASE broadcasts of previous rounds, it computes the new paths, updates the variables *seq* and *path*, updates its routing table entries to reflect the new path and reverses the direction, in G_r, of edges which are on the new path.

Algorithm for the source node S
Parameters: A bound τ on the round-trip delay between source and destination, and a bound δ on the end-to-end propogation delay of a broadcast message.
Input: A list of neighbors in L and the desired number k of disjoint paths
Output: Up to k routing table entries corresponding k disjoint paths from S to T

$num\_paths\_known := 0$; $seq := 0$; $path := \emptyset$
while (true) {
  if ($num\_paths\_known < k$)
    $seq_{PDT} := seq + 1$
    send a copy of PDT$\langle S, T, seq_{PDT}\rangle$ to all neighbors in G_r
    wait for PDT_{Ack} broadcast with $\text{PDT}_{\text{Ack}}.seq = seq_{PDT}$ from T for τ units of time
      meanwhile ignore PDTs for $\langle S, T\rangle$
    if PDT_{Ack} with $\text{PDT}_{\text{Ack}}.seq = seq_{PDT}$ arrives
      $seq := seq + 1$; broadcast PMT
      $num\_paths\_known := num\_paths\_known + 1$
      compute the new set of paths and update *path*
      find successor h from PMT.*path* and put H as the next hop in the routing table entry $\langle S, T\rangle$
    else $seq := seq + 1$; broadcast PMT$\langle null, seq\rangle$
    wait for δ units of time so that the PMT broadcast has reached every node
  if (($num\_paths\_known = k$) and LF is empty) wait until there is an entry in LF

for every entry in LF
if the link failure affects one of the current paths (stored in $path$)
$seq := seq + 1$; broadcast PATH_ERASE
$num\_paths\_known := num\_paths\_known - 1$
remove the corresponding path from $path$
remove corresponding next hop in the routing table for $\langle S, T \rangle$ entry
wait for δ units of time so that the PATH_ERASE broadcast has reached every node
remove entry from LF
}

Algorithm for the destination node T
Input: A list of neighbors in L
when a PDT arrives with a sequence number seq
if T has already broadcast PDT_{Ack} for seq, then ignore the PDT
else broadcast PDT_{Ack}

Algorithm for the intermediate node i
Input: A list of neighbors in L
Output: Next hop entry in the routing table for $\langle S, T \rangle$
initially, $seq := 0$; $seq_{PDT} := 0$; $path := \emptyset$
when a PDT arrives
if PDT.seq $> seq_{PDT}$ and PDT.$seq = seq + 1$
append i to the PDT.$path$ and send a copy of the modified PDT to all neighbors in G_r
seq_{PDT} := PDT.seq
else if PDT.$seq > seq_{PDT}$ store it in FUTURE array
else discard it
when a PMT arrives
if PMT.$seq = seq + 1$
$seq := seq + 1$; compute the new set of paths and update $path$
if i is on PMT.$path$
find successor h and predecessor p from the PMT.$path$
L[h].direction := in //reverse edge $\langle i, h \rangle$
L[p].direction := out //reverse edge $\langle p, i \rangle$
put H as the next hop in I's routing table for $\langle S, T \rangle$ entry
if there are any entries in the FUTURE array which can now be processed
process them and remove them from the array
else store it in FUTURE array and request neighbors for missing previous broadcasts
when a PATH_ERASE arrives
if PATH_ERASE.$seq = seq + 1$
$seq := seq + 1$; remove the corresponding path from $path$
if i is on PATH_ERASE.$path$
find successor h and predecessor p from the PATH_ERASE.$path$
L[h].direction := out //reverse edge $\langle i, h \rangle$
L[p].direction := in //reverse edge $\langle p, i \rangle$
remove H as the next hop in I's routing table for $\langle S, T \rangle$ entry
if there are any entries in the FUTURE array which can now be processed
process them and remove them from the array
else store it in FUTURE array and request neighbors for missing previous broadcasts
when a neighboring link failure is discovered broadcast a LINK_FAIL message

6 Conclusion

This paper presents simple, distributed algorithms for the well-studied problem of finding disjoint paths in networks. This problem has taken on new importance in light of its potential application to the problem of routing in mobile networks, where maintaining route information under topological changes is challenging. Our next step is to experimentally evaluate the performance of the proposed algorithms.

In contrast to much of the empirical work that was proposed in recent years, our algorithm is a simple distributed implementation of a proven sequential method. Although, in theory, the set of disjoint paths found might not contain the shortest path, in practice our breadth-first search based algorithms will likely result in paths with small link delays.

The messages used in algorithms in Section 4 are of a small fixed length. In case of the resilient version, the size of messages during path computation is proportional to the path length. However, unlike in source routing, the data packets need not carry path information. Our algorithm discovers paths one at a time. The paths that have been found can be used for communication while others are being discovered.

References

[CDS98] J. Chen, P. Druschel, and D. Subramanian. An Efficient Multipath Forwarding Method. In *Proceedings of INFOCOM-98*, pages 1418–1425, 1998.

[CKG90] C. Cheng, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A Distributed Algorithm for Finding K Disjoint Paths of Minimum Total Length. In *Proceeding of 28th Annual Allerton Conference on Communication, Control, and Computing*, Urbana, Illinois, October, 1990.

[C89] R. Coltun. OSPF: An Internet Routing Protocol. *ConneXions*, 3 (8):19–25, 1989.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Ed., MIT Press, Cambridge, 2001.

[H88] C. Hedrick. Routing Information Protocol. Internet Request for Comment (RFC) 1058, 1988.

[H00] C. Huitema. *Routing in the Internet*, 2nd Ed., Prentice Hall PTR, 2000.

[IR84] A. Itai and M. Rodeh. The Multi-Tree Approach to Reliability in Distributed Networks. In *Proceedings of 25th FOCS*, pages 137–147, 1984.

[LG01] S. J. Lee and M. Gerla. Split Multipath Routing With Maximally Disjoint Paths in Ad Hoc Networks. In *Proceedings of the IEEE ICC*, pages 3201–3205, 2001.

[NCD01] A. Nasipuri, R. Castaneda, and S. R. Das. Performance of Multipath Routing for On-Demand Protocols in Ad Hoc Networks. *ACM/Kluwer Mobile Networks and Applications (MONET) Journal*, Vol. 6, No. 4, pages 339–349, 2001.

[ORS93] R. Ogier, V. Rutenburg, and N. Shacham. Distributed Algorithms for Computing Shortest Pairs of Disjoint Paths. *IEEE Trans. Information Theory*, pages 443–455, March, 1993.

[PHS02] P. Papadimitratos, Z. J. Haas, and E. G. Sirer. Path Set Selection in Mobile Ad Hoc Networks. In *Proceeding of the ACM MOBIHOC*, pages 160–170, 2002.

[PHST00] M. R. Pearlman, Z. J. Haas, P. Sholander, and S. S. Tabrizi. On the Impact of Alternate Path Routing for Load Balancing in Mobile Ad Hoc Networks. In *Proceeding of the ACM MOBIHOC*, pages 3–10, 2000.

[SNA91] D. Sidhu, R. Nair, and S. Abdallah. Finding Disjoint Paths in Networks. In *Proceedings of SIGCOMM-91*, pages 43–51, 1991.

[VSR03] A. Valera, W. Seah, and S. V. Rao. Cooperative Packet Caching and Shortest Multipath Routing in Mobile Ad hoc Networks. In *Proceeding of the IEEE INFOCOM*, 2003.

[VG01] S. Vutukury and J. J. Garcia-Luna-Aceves. MDVA: A Distance-Vector Multipath Routing Protocol. In *Proceedings of the IEEE INFOCOM*, pages 557–564, 2001.

[WH01] K. Wu and J. Harms. Performance Study of a Multipath Routing Method for Wireless Mobile Ad Hoc Networks. In *Proceedings of the IEEE Int'l Symposium on Modeling, Analysis and Simulation of Compute and Telecommunication Systems (MASCOTS)*, pages 99–107, 2001.

[YKT03] Z. Ye, S. V. Krishnamurthy, and S. K. Tripathi. A Framework For Reliable Routing in Mobile Ad Hoc Networks. In *Proceeding of the IEEE INFOCOM*, 2003.