Efficient Algorithms for Single Link Failure Recovery and Its Application to ATM Networks

AMIT M. BHOSLE and TEOFILO F. GONZALEZ
Department of Computer Science
University of California
Santa Barbara, CA 93106
{bhosle,teo}@cs.ucsb.edu

ABSTRACT
We investigate the single link failure recovery problem and its application to the alternate path routing problem for ATM networks. Specifically, given a 2-connected graph G, a specified node s, and a shortest paths tree $\mathcal{T}_s = \{e_1, e_2, \ldots, e_{n-1}\}$ of s, where $e_i = (x_i, y_i)$ and $x_i = parent_{\mathcal{T}_s}(y_i)$, find a shortest path from y_i to s in the graph $G \backslash e_i$ for $1 \leq i \leq n-1$. We present an $O(m + n \log n)$ time algorithm for this problem and a linear time algorithm for the case when all weights are equal. When the edge weights are integers, we present an algorithm that takes $O(m + T_{sort}(n))$ time where $T_{sort}(n)$ is the time required to sort n integers. We show that any solution to the single link recovery problem can adapted to solve the alternate path routing problem in ATM networks.

KEY WORDS
Single Link Failure Recovery, ATM Networks, Efficient Algorithms, Alternate Path Routing.

1 Introduction

The graph G represents a set of nodes in a network and the weight of the link represent the cost (say time) for transmitting a message through the link. The shortest path tree $\mathcal{T}_s$ specifies the best way of transmitting to node s a message originating at any given node in the graph. When the links in the network have transient faults, we need to find a way to recover from such faults. In this paper we consider the case when there is only one link failure, the failure is transient, and information about the failure is not propagated throughout the network. That is, a message originating at node x with destination s will be sent along the path specified by $\mathcal{T}_s$ until it reaches node s or a link that failed. In the latter case, we need to use a shortest recovery path to s from that point. Since we assume single link faults and the graph is 2-connected, such a path always exist. We call this problem the Single Link Failure Recovery (SLFR) problem. As we show later on, this problem has applications to the Alternate Path Routing (APR) problem for ATM networks. The SLFR problem has applications when there is no global knowledge of a link failure, in which case the failure is discovered only when one is about to use the failed link. In such cases the best option is to take a shortest path from the point one discovers the failure to the destination avoiding the failed link.

A naive algorithm for the SLFR problem is based on recomputation. For every edge $e_i = (x_i, y_i)$ in the shortest path tree $\mathcal{T}_s$, compute the shortest path from y_i to s in the graph $G \backslash e_i$. This algorithm requires $n-1$ invocations of the single source shortest path algorithm. An implementation of Dijkstra's algorithm that uses Fredman and Tarjan's Fibonacci Heaps takes $O(m + n \log n)$ time, which currently it is the fastest single source shortest paths algorithm. The overall time complexity of the naive algorithm is thus $O(mn + n^2 \log n)$. This naive algorithm also works for the directed version of the SLFR problem.

One of the main applications of our work is the *alternate path routing* (APR) problem for ATM networks. This problem arises when using the Interim Inter-switch Signaling Protocol (IISP)[4]. This protocol has been implemented by ATM equipment vendors as a simple interim routing solution for the dynamic routing mechanism given by the Private Network-Network Interface (PNNI)[3]. IISP is sometimes referred to as PNNI(0) and provides the network basic functionality for path selection at setup time. Assuming correct primary routing tables, the protocol implements a depth-first search mechanism using the alternate paths when the primary path leads to dead-ends due to link failure. Routes disconnected by a link failure can be reestablished along the alternate path.

IISP does not propagate link failure information. Newer protocols, like PNNI, can find new paths and adapt automatically when links fail. However that process is CPU intensive and is not desirable when only transient failures occur, which is the scenario that we consider in this paper. Additional IISP details are given in [11].

A solution to the SLFR problem is not a solution to the APR problem. However, we show how to obtain a solution to the APR problem from any solution to the SLFR problem. Configuring the primary and alternate path tables should be in such a way that reachability under single link failures is ensured while maintaining, to a limited extent, shortest recovery paths. This is a non-trivial task and the normal practice is to perform them manually. Slosiar and Latin [11] studied this problem and presented an $O(n^3)$

time algorithm. Since there can be $O(n)$ destinations in the network, their algorithm takes $O(n^4)$ time in the worst case.

1.1 Main Results

Our main results are (near) optimal algorithms for single link failure recovery (SLFR) problem, and (near) optimal algorithms for the alternate path routing (APR) problem. Specifically, we present an $O(m + n \log n)$ time algorithm for SLFR problem. We present an $O(m + n)$ time algorithm for the case when all the edge weights are same. When the edge weights are integers, we present an algorithm that takes $O(m + T_{sort}(n))$ time, where $T_{sort}(n)$ is the time required to sort n integers. Currently, $T_{sort}(n) = O(n \log \log n)$ due to Han [8]. The computation of the shortest path tree can also be included in all the above bounds, but for simplicity we say the the shortest path tree is part of the input to the problem. We show in Section 5 that all of the above algorithms can be adapted to the alternate path routing (APR) problem within the same time complexity bounds by showing that in linear time one may transform any solution to the SLFR problem to the APR problem.

1.2 Preliminaries

Our communication network is modeled by a weighted undirected 2-connected graph $G(V, E)$, with $n = |V|$ and $m = |E|$. Each edge $e \in E$ has an associated cost, $cost(e)$, which is a non-negative real number. We use $path_G(s, t)$ to denote the shortest path between s and t in graph G and $d_G(s, t)$ to denote its weight. A cut in a graph is the partitioning of the set of vertices V into V_1 and V_2 and it is denoted by (V_1, V_2). The set of edges $E(V_1, V_2)$ represents all the edges across the cut (V_1, V_2).

Finally, a shortest path tree $\mathcal{T}_s$ for a node s is a collection of $n - 1$ edges of G such that $\{e_1, e_2, \ldots, e_{n-1}\}$ where $e_i = (x_i, y_i)$, $x_i, y_i \in V$, $x_i = parent_{\mathcal{T}_s}(y_i)$ and the path from node v to s in $\mathcal{T}_s$ is a shortest path from v to s in G. We remove the index $\mathcal{T}_s$ from $parent$ when it is clear the tree $\mathcal{T}_s$ we mean. Note that under our notation a node $v \in G$ is the x_i component of as many tuples as the number of its children in $\mathcal{T}_s$ and it is the y_i component in one tuple (if $v \neq s$). Nevertheless, this notation facilitates an easier formulation of the problem. Moreover, our algorithm does not depend on this labeling.

2 A Simple $O(m \log n)$ Algorithm

In this section we describe a simple algorithm for the SLFR problem which runs in $O(m \log n)$ time and in Section 3 we use it to derive an algorithm that takes $O(m + n \log n)$ time.

When the edge $e_i = (x_i, y_i)$ of the shortest path tree $\mathcal{T}_s$ is deleted, $\mathcal{T}_s$ is split into two components. Let us denote the component containing s by $V_{s|i}$ and the other by V_i. Consider the cut $(V_{s|i}, V_i)$ in G. Among the edges crossing this cut, only one belongs to $\mathcal{T}_s$, namely $e_i = (x_i, y_i)$. Since G is 2-connected, we know that there is at least one non-tree edge in G that crosses the cut. Our algorithm is based on the following lemma that establishes the existence of a shortest path from y_i to s in the graph $G \backslash e_i$ that uses exactly on edge of the cut $(V_i, V_{s|i})$. For brevity we do not include the proof.

Lemma 2.1 *There exists a shortest path from y_i to s in the graph $G \backslash \{e_i = (x_i, y_i)\}$ that uses exactly one edge of the cut $(V_i, V_{s|i})$ and its weight is equal to*

$$d_{G \backslash e_i}(y_i, s) = MIN_{(u,v) \in E(V_i, V_{s|i})}\{weight(u, v)\} \quad (1)$$

where $(u, v) \in E(V_i, V_{s|i})$ signifies that $u \in V_i$ and $v \in V_{s|i}$ and the weight associated with the edge (u, v) is given by

$$weight(u, v) = d_G(y_i, u) + cost(u, v) + d_G(v, s) \quad (2)$$

The above lemma immediately suggests an algorithm for the SLFR problem. From each possible cut, select an edge satisfying equation (1). An arbitrary way of doing this may not yield any improvement over the naive algorithm since there may be as many as $\Omega(m)$ edges across each of the $n - 1$ cuts to be considered, leading to $\Omega(mn)$ time complexity. However, an ordered way of computing the recovery paths enables us to avoid this $\Omega(mn)$ bottleneck.

Our problem is reduced to mapping each edge $e_i \in \mathcal{T}_s$ to an edge $a_i \in G \backslash \mathcal{T}_s$ such that a_i is the edge with minimum weight in $E(V_i, V_{s|i})$. We call a_i the *escape edge* for e_i and use $\mathcal{A}$ to denote this mapping function. We replace equation (1) with the following equation to compute $\mathcal{A}(e_i)$.

$$\mathcal{A}(e_i) = a_i \iff weight(a_i) = MIN_{(v,u) \in E(V_{s|i}, V_i)}\{weight(u, v)\} \quad (3)$$

Once we have figured out the escape edge a_i for each e_i, we have enough information to construct the required shortest recovery path.

The *weight* as specified in equation (2) for the edges involved in the equation (3) depends on the deleted edge e_i. This implies additional work for updating these values as we move from one cut to another, even if the edges across the two cuts are the same. Interestingly, when investigating the edges across the cut $(V_i, V_{s|i})$ for computing the escape edge for the edge $e_i = (x_i, y_i)$, if we add the quantity $d(s, y_i)$ to all the terms involved in the minimization expression, the minimum weight edge retrieved remains unchanged. However, we get an improved weight function. The weight associated with an edge (u, v) across the cut is now defined as:

$$\begin{aligned} & weight(u, v) \\ &= d_G(s, y_i) + d_G(y_i, u) + cost(u, v) + d_G(v, s) \\ &= d_G(s, u) + cost(u, v) + d_G(v, s) \end{aligned} \quad (4)$$

Now the weight associated with an edge is independent of the cut being considered and we just have to design an efficient method to construct the set $E(V_i, V_{s|i})$ for all i.

2.1 Description of the Algorithm

We employ a bottom-up strategy for computing the recovery paths. None of the edges of $\mathcal{T}_s$ would appear as an escape edge for any other tree edge because no edge of $\mathcal{T}_s$ crosses the cut induced by the deletion of any other edge of $\mathcal{T}_s$. As the first step of the algorithm, we construct n heaps, one for each node in G. The heaps contain elements of the form $< e, weight(e) >$ where e is a non-tree edge with $weight(e)$ as specified by equation (4). The heaps are maintained as *min heaps* according to the $weight(\cdot)$ values of the edges in it. Initially the heap H_v corresponding to the node v contains an entry for each non-tree edge in G adjacent to v. When v is a leaf in $\mathcal{T}_s$, H_v contains all the edges crossing the cut induced by deleting the edge (u, v) where $u = parent_{\mathcal{T}_s}(v)$ is the parent of v in $\mathcal{T}_s$. Thus, the recovery path for the leaf nodes can be easily computed at this time by performing a *findMin* operation on the corresponding heaps.

Let us now consider an internal node v whose children in $\mathcal{T}_s$ have had their recovery paths computed. Let the children of v be the nodes $v_1, v_2, \ldots, v_k$. The heap for node v is updated as follows:

$$H_v \leftarrow meld(H_v, H_{v_1}, H_{v_2}, \ldots, H_{v_k})$$

Now H_v contains all the edges crossing the cut induced by deleting the edge $(parent_{\mathcal{T}_s}(v), v)$. But it also contains other edges which are completely contained inside V_v which is the set of nodes in the subtree of $\mathcal{T}_s$ rooted at v. However, if e is the edge retrieved by the $findMin(H_v)$ operation, after an initial linear time preprocessing, we can determine in constant time whether or not e is an edge across the cut. The preprocessing begins with a DFS (depth first search) labeling of the tree $\mathcal{T}_s$. Each node v needs an additional integer field, which we call min, to record the smallest DFS label for any node in V_v. It follows from the property of dfs-labeling that an edge $e = (a, b)$ is not an edge crossing the cut if and only if $v.min < dfs(a) < dfs(v)$ and $v.min < dfs(b) < dfs(v)$. In case e happens to be an *invalid* edge (i.e. an edge not crossing the cut), we perform a $deleteMin(H_v)$ operation. We continue performing the $findMin(H_v)$ followed by $deleteMin(H_v)$ operations until $findMin(H_v)$ returns a *valid* edge.

The analysis of the above algorithm is straightforward and its time complexity is dominated by the heap operations involved. Using F-Heaps, we can perform the operations $findMin$, $insert$ and $meld$ in amortized constant time, while $deleteMin$ requires $O(\log n)$ amortized time. The overall time complexity of the algorithm can be shown to be $O(m \log n)$. We have thus established the following theorem whose proof is omitted for brevity.

Theorem 2.1 *Given an undirected weighted graph $G(V, E)$ and a specified node s, the shortest and the recovery paths from all nodes to s is computed by our procedure in $O(m \log n)$ time.*

3 A Near Optimal Algorithm

We now present a near optimal algorithm for the SLFR problem which takes $O(m + n \log n)$ time to compute the recovery paths to s from all the nodes of G. The key idea of the algorithm is based on the following observation: If we can compute a set $E_{\mathcal{A}}$ of $O(n)$ edges which includes all those which can possibly figure as an escape edge a_i for any edge $e_i \in \mathcal{T}_s$ and then invoke the algorithm presented in the previous section on $G(V, E_{\mathcal{A}})$, we can solve the entire problem in $O(T_p(m, n) + n \log n)$ time, where $T_p(m, n)$ is the preprocessing time required to compute the set $E_{\mathcal{A}}$. We now show that $E_{\mathcal{A}}$ can be computed in $O(m + n \log n)$ time, thus solving the problem in $O(m + n \log n)$ time.

Recall that to find the escape edge for $e_i \in \mathcal{T}_s$ we need to find the minimum weighted edge across the induced cut $(V_i, V_{s|i})$ where the weight of an edge is as defined in equation (4). This objective reminds us of *minimum cost spanning trees* since they contain the lightest edge across any cut. The following *cycle property* about MSTs is folklore and we state it without proof:

Property [MST]: *If the heaviest edge in any cycle in a graph G is unique, it cannot be part of the minimum cost spanning tree of G.*

Computation of $E_{\mathcal{A}}$ is now intuitive. We construct a weighted graph $G_{\mathcal{A}}(V, E^{\mathcal{A}})$ from the input graph $G(V, E)$ as follows: $E^{\mathcal{A}} = E \backslash E(\mathcal{T}_s)$, where $E(\mathcal{T}_s)$ are the edges of $\mathcal{T}_s$, and the weight of edge $(u, v) \in E^{\mathcal{A}}$ is defined as in Equation (4), i.e, $weight(u, v) = d_G(s, u) + cost(u, v) + d_G(v, s)$.

Note that the graph $G_{\mathcal{A}}(V, E^{\mathcal{A}})$ may be disconnected because we have deleted $n-1$ edges from G. Next, we construct a minimum cost spanning forest of $G_{\mathcal{A}}(V, E^{\mathcal{A}})$. A minimum cost spanning forest for a disconnected graph can be constructed by finding a minimum cost spanning tree for each component of the graph. The minimum cost spanning tree problem has been extensively studied and there are well known efficient algorithms for it. Using F-Heaps, Prim's algorithm can be implemented in $O(m + n \log n)$ time for arbitrarily weighted graphs [6]. The problem also admits linear time algorithms when edge weights are integers [5]. Improved algorithms are given in [10, 2, 6]. The set $E_{\mathcal{A}}$ contains precisely the edges present in the minimum cost spanning forest (MSF) of $G_{\mathcal{A}}$. The following lemma, which for brevity we omit its proof, establishes that $E_{\mathcal{A}}$ contains all the candidate escape edges a_i.

Lemma 3.1 *For any edge $e_i \in \mathcal{T}_s$, if $\mathcal{A}(e_i)$ is unique, it has to be an edge of the minimum spanning forest of $G_{\mathcal{A}}$. If $\mathcal{A}(e_i)$ is not unique, a minimum spanning forest edge offers*

a recovery path of the same weight.

It follows from Lemma 3.1 that we need to investigate only the edges present in the set $E_{\mathcal{A}}$ as constructed above. Also, since $E_{\mathcal{A}}$ is the set of edges of the MSF, (1) $|E_{\mathcal{A}}| \leq n-1$ and (2) for every cut (V, V') in G, there is at least one edge in $E_{\mathcal{A}}$ crossing this cut. We now invoke the algorithm presented in Section 2 which requires only $O((|E_{\mathcal{A}}|+n)\log n)$ which is $O(n \log n)$ additional time to compute all the required recovery paths. The overall complexity of our algorithm is thus $O(m+n\log n)$ time which is includes the constructions of the shortest paths tree of s in G and the minimum spanning forest of $G_{\mathcal{A}}$ required to compute $E_{\mathcal{A}}$. We have thus established the following theorem.

Theorem 3.1 *Given an undirected weighted graph $G(V,E)$ and a specified node s, the shortest and the recovery paths from all nodes to s is computed by our procedure in $O(m+n\log n)$ time.*

4 Unweighted Graphs

In this section we present a linear time algorithm for the *unweighted* SLFR, thus improving the $O(m+n\log n)$ algorithm of Section 3 for this special case. One may view an unweighted graph as a weighted one with all edges having unit cost. As in the arbitrarily weighted version, we assign each non-tree edge a new weight as specified by equation (4). The recovery paths are determined by considering the non-tree edges from smallest to largest (according to their new weight) and finding the nodes for which each of them can be an escape edge. The algorithm, `S-L`, is given below.

```
Procedure S-L(v)
  Sort the non-tree edges by their weight;
  for each non-tree edge e = (u, v) in ascend order do
    Let w be the nearest common ancestor of u and v in T_s
    The recovery path for all the nodes lying on
      path_{T_s}(u, w) and path_{T_s}(v, w) including u and v,
      but excluding w that have their recovery paths
      undefined are set to use the escape edge e;
  endfor
End Procedure S-L
```

The basis of the entire algorithm can be stated in the following lemma. We omit the proof from this extended abstract.

Lemma 4.1 *If $e = (u,v) = deleteMin(L).edge$, and $w = nca(u,v)$ is the* nearest common ancestor *of u and v in $\mathcal{T}_s$, the recovery paths for all the nodes lying on $path_{\mathcal{T}_s}(u,w)$ and $path_{\mathcal{T}_s}(v,w)$ including u and v but excluding w, whose recovery paths have not yet been discovered, use the escape edge e.*

4.1 Implementation Issues

Since any simple path in the graph can have at most $n-1$ edges, the newly assigned weights of the non-tree edges are integers in the range $[1, 2n]$. As the first step, we sort these non-tree edges according to their weights in linear time. Any standard algorithm for sorting integers in a small range can be used for this purpose. E.g. *Radix sort* of n integers in the range $[1,k]$ takes $O(n+k)$ time. The sorting procedure takes $O(m+n)$ time in this case. This set of sorted edges is maintained as a linked list, L, supporting *deleteMin* in $O(1)$ time where $deleteMin(L)$ returns and deletes the smallest element present in L.

Linear time algorithms for the *nearest common ancestor* are given in [9, 1]. Using these algorithms, after a linear time preprocessing, in constant time one can find the nearest common ancestor of any two specified nodes in a given tree.

Our algorithm uses efficient *Union-Find* structures. Several fast algorithms for the general union-find problem are known, the fastest among which runs in $O(n + m\alpha(m+n,n))$ time and $O(n)$ space for executing an intermixed sequence of m union-find operations on an n-element universe [12], where α is the functional inverse of Ackermann's function. Although the general problem has a super-linear lower bound [13], a special case of the problem admits linear time algorithm [7]. The requirements for this special case are that the "union-tree" has to be known in advance and the only union operations, which are referred as "unite" operations, allowed are of the type $unite(parent(v), v)$ where $parent(v)$ is the parent of v in the "union-tree". The reader is referred to [7] for the details of the algorithm and its analysis. As we shall see, the union-find operations required by our algorithm fall into the set of operations allowed in [7] and we use this linear time union-find algorithm. With regard to the running time, our algorithm involves $O(m)$ $find(\cdot)$ and $\Theta(n)$ $union(\cdot)$ operations on an n- element universe, which take $O(m+n)$ total time.

Our algorithm, `All-S-L`, is formally described below.

```
Procedure All-S-L
  Preprocess T_s using a linear time algorithm [1, 9] to
    answer quickly the nearest common ancestor queries.
  Initialize the union-find data-structure of [7].
  Assign weights to the non-tree edges as specified by
    equation (4) and sort them by these weights.
    Store the sorted edges in a priority queue structure L,
    supporting deleteMin(L) in O(1) time.
  Mark node s and unmark all the remaining nodes.
  while there is an unmarked vertex do
    {e = (u, v)} = deleteMin(L).edge;
    w = nca(u, v);
    for x = u, v do
      if x is marked then x = find(x); endif
    endfor
    while (find(x) ≠ find(w)) do
      A(parent(x), x) = e;
      union(find(parent(x)), find(x));
      Mark x;
      x = parent(x);
```

```
   endwhile
  endwhile
End Procedure All-S-L
```

Correctness follows from the fact the that procedure `All-S-L` just implements procedure `S-L` and Lemma 4.1 shows that the strategy followed by procedure `S-L` generates recovery paths for all the nodes in the graph. The time complexity is linear, following the discussion before the procedure. We have thus established the following theorem.

Theorem 4.1 *Given an undirected unweighted graph $G(V, E)$ and a specified node s, the shortest and the recovery paths from all nodes to s is computed by our procedure in $O(m + n)$ time.*

5 Alternate Paths Routing for ATM Networks

Let us now discuss some details about the IISP protocol for ATMs. Whenever a node receives a message it receives the tuple $[(s)(m)(l)]$. where s is the final destination for the message, m is the message being sent and l is the last link traversed. Each node has two tables: primary and alternate. The primary table gives for every destination node s the next link to be taken. When a link x fails, then the primary table entries that contain x as the next link are automatically deleted and when the link x becomes available all the original entries in the table that contained that link are restored. The alternate path table contains a link to be taken when either there is no entry for the destination s, or when the last link is the same as the link for s in the primary table. The alternate table provides a mechanism to recover from link failures.
For the purpose of this paper the ATM routing mechanism operates as follows.

```
Routing Protocol(p)
  Protocol is executed when node p receives the tuple
    [ (s: destination) (m: message) (l: last link) ]
  if p = s then node s has received the message; exit;
  endif
  let q be the next link in the primary path for s (info taken
    from the primary table)
  case
    : q is void or q = last link:
      send (destination s) (message) through the link in the
        alternate table for entry s;
    : q ≠ l: send (destination s) (message) through q
  endcase
End Routing Protocol
```

The primary routing tables for each destination node s is established by constructing a shortest path tree rooted at s. For every node x in the tree the path from x to s is a shortest path in the graph (or network). So the primary routing table for node x contains $parent\mathcal{T}_s(x)$ in its entry for s.

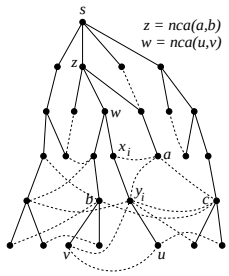

Figure 1. Recovery paths in undirected unweighted graphs.

The alternate path routing problem for ATM networks consists of generating the primary and alternate routing tables for each destination s. The primary routing table is defined in the previous paragraph. The entries int the alternate tables are defined for the alternate path routes. These paths are defined as follows. Consider the edge $e_i = (x_i, y_i)$ and $x_i = parent_{\mathcal{T}_s}(y_i)$. The alternate path route for edge e_i is the escape edge $e = (u, v)$ with v a descendent of y_i in the tree $\mathcal{T}_s$ if an ancestor of y_i in tree $\mathcal{T}_s$ has e as its escape edge. Otherwise, it is computed as in Equation (4). This definition of the problem is given in [11].

In this section we describe a linear time post-processing to generate, from a solution to the SLFR problem, a set of alternate paths which ensure loop-free connectivity under single link failures in ATM networks. While the set of alternate paths generated by the algorithm in Section 3 ensure connectivity, they may introduce loops since the IISP [4] mechanism does not have the information about the failed edge, it cannot make decisions based on the failed edge. Thus, we need to ensure that each router has a *unique* alternate path entry in its table. For example in Figure 2, it is possible that $\mathcal{A}(w, x_i) = (y_i, a)$ and $\mathcal{A}(s, z) = (y_i, c)$. Thus, y_i needs to store two entries for alternate paths depending on the failed edge. In this particular case, y_i should preferably store the entry (y_i, c) since it provides loop-free connectivity even when (w, x_i) fails (though possibly sub-optimal). Contrary to what was stated in [11], storing at most one alternate entry per node does not ensure loop-free routing. E.g. If $\mathcal{A}(w, x_i) = (y_i, a)$ and $\mathcal{A}(s, z) = (x_i, a)$, and (s, z) fails, x_i routes the traffic via a, instead of forwarding it to y_i, thus creating a loop. We need to ensure that for all $e \in path_{\mathcal{T}_s}(y_i, s)$, $\mathcal{A}(e) = (y_i, c)$. This is the key to the required post-processing which retains the desirable set of alternate paths from the set of paths generated so far. We formally describe our post-processing algorithm below.

Algorithm Generate Loop-free Alternate Paths (`GLAP`) takes as global parameters a shortest path tree $\mathcal{T}_s$ and the escape edge for each edge, e, $\mathcal{A}(e)$ and it generates alternate path routes as defined above. The procedure has as input a node r in $\mathcal{T}_s$. Initially every node is unmarked

and procedure GLAP is invoked with GLAP(s).

```
Procedure GLAP( r )
  for every node z ∈ T_s such that z = child_{T_s}(s), and
        z is not marked do
    (b, c) = A(r, z) such that b ∈ V_z (where V_z is the set
      of vertices in the subtree of T_s
      rooted at z)
    while (b ≠ z) do
      A(parent_{T_s}(b), b) = (b, c)
      Mark b
      GLAP(b)
      b = parent_{T_s}(b)
    endwhile
  endfor
End Procedure GLAP
```

The $O(n)$ time complexity comes from the fact that any edge of $\mathcal{T}_s$ is investigated at most twice. The while loop takes care that all edges on $path_{\mathcal{T}_s}(z, b)$ are assigned (b, c) as their alternate edge. The recursive calls update the alternate edges of the edges that branch off from $path_{\mathcal{T}_s}(z, b)$ while the main for loop makes sure that all paths branching off from the source node s are investigated.

Theorem 5.1 *Given a solution to the SLFR problem for s tree of shortest paths $\mathcal{T}_s$, our procedure constructs a solution to the alternate path routing problem for ATM networks in $O(n)$ time.*

6 Integer Edge Weights SLFR

If the edge weights are integers, linear time algorithms are known for the shortest paths tree [14] and the minimum cost spanning tree [5]. We can reduce the number of edges to be sorted from $O(m)$ to $O(n)$ using the technique of investigating only the MST edges. After sorting the edges in $T_{sort}(n)$ time, we use the algorithm for unweighted graphs to solve the problem in $O(n)$ additional time. Currently $T_{sort}(n) = O(n \log\log n)$ due to Han [8]. We have thus established the following theorem.

Theorem 6.1 *Given an undirected graph $G(V, E)$ with integer edge weights, and a specified node s, the shortest and the recovery paths from all nodes to s can be computed by our procedure in $O(m + T_{sort}(n))$ time.*

7 Concluding Remarks

In this paper we have presented near optimal algorithms for the undirected version of the SLFR problem. For *directed acyclic graphs*, the problem admits a linear time algorithm. This is because in a DAG, a node v *cannot* have any edges directed towards any node in the subtree of $\mathcal{T}_s$ rooted at v (since this would create a cycle). Thus, we only need to minimize over $\{cost(v, u) + d_G(u, s)\}$ for all $(v, u) \in E$ and $u \neq parent_{\mathcal{T}_s}(v)$, to compute the recovery path from v to s since $path_G(u, s)$ cannot contain the failed edge $(parent_{\mathcal{T}_s}(v), v)$ and remains intact on its deletion. We thus need only $\sum_{v \in V}(out\_degree(v)) = O(m)$ additions/comparisons to compute the recovery paths.

References

[1] A.L. Buchsbaum, H. Kaplan, A. Rogers, and J.R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. In *Proc. 30th STOC*, pages 279-288. ACM Press, 1998.

[2] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM*, 47:1028-1047, 2000.

[3] ATM Forum. 94-0471r16. *PNNI Routing Specification.*

[4] ATM Forum. af-pnni-0026.000. *Interim Inter-switch Signalling Protocol (IISP) Specification v1.0*, 1996.

[5] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *JCSS*, 48:533-551, 1994.

[6] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34:596-615, 1987.

[7] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *JCSS*, 30(2):209-221, 1985.

[8] Y. Han. Deterministic sorting in O$(n \log\log n)$ time and linear space. In *Proc. 34th STOC*, pages 602–608. ACM Press, 2002.

[9] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SICOMP, 13(2)*, pages 338-355, 1984.

[10] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. In *Automata, Languages and Programming*, pages 49-60, 2000.

[11] R. Slosiar and D. Latin. A polynomial-time algorithm for the establishment of primary and al ternate paths in atm networks. In *Proc. of IEEE INFOCOM*, pages 509-518, 2000.

[12] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22(2):215-225, 1975.

[13] R.E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *JCSS*, 18(2):110-127, 1979.

[14] M. Thorup. Undirected single source shortest path in linear time. In *FOCS*, pages 12–21, 1997.