Algorithms for Single Link Failure Recovery and Related Problems

Amit M. Bhosle* 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, and the k-replacement edges for each edge of a minimum cost spanning tree. 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 establish a lower bound of $\Omega(min(m\sqrt{n}, n^2))$ for the directed[1] *version of our problem under the path comparison model. We show that any solution to the single link recovery problem can be adapted to solve the alternate path routing problem in ATM networks. Our technique to solve the single link failure recovery problem is adapted to find the k-replacement edges for the tree edges of a minimum cost spanning tree in $O(m + n \log n)$ time.*

KEY WORDS: Single Link Failure Recovery, ATM Networks, Efficient Algorithms, Alternate Path Routing, MST k-replacement edges, MST k most vital edges.

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 may be susceptible to 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 exists. 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

*Currently at YASU Tech. Pvt. Ltd., Hyderabad - 500 016, India.

[1]In the directed version, $\mathcal{T}_s$ is the shortest paths *destination* tree of s.

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) [6]. 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) [5]. 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 re-established 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 [20].

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 [20] studied this problem and presented an $O(n^3)$ time algorithm.

A closely related problem is that of finding the *replacement edges* for the *tree* edges of the *minimum cost spanning tree* $\mathcal{T}_{mst}$ of a given graph. Formally, given a weighted undirected graph $G(V, E)$, and the minimum weight (cost) spanning tree, $\mathcal{T}_{mst}$, of G, find for each edge $e_i \in \mathcal{T}_{mst}$ the minimum cost edge of $E \setminus e_i$ which connects the two disconnected components of $\mathcal{T}_{mst} \setminus e_i$. Efficient algorithms for this problem have been presented in [23, 4]. A straight forward generalization of this problem, termed `k-RE-MST`, is defined as follows:

`k-RE-MST`: Given an undirected weighted graph $G(V, E)$, the minimum weight (cost) spanning tree $\mathcal{T}_{mst}$ of G, and an integer k, for each edge $e \in \mathcal{T}_{mst}$, find the k least cost edges (in order of increasing weight) across the cut induced by deleting e from $\mathcal{T}_{mst}$.

We assume that the graph is k edge connected and that k is a constant. The `k-RE-MST` problem was introduced by Shen [19] as a subproblem in a randomized algorithm for the *k most vital edges (k-MVE) with respect to a given minimum cost spanning tree* problem. Shen's randomized algorithm has a time complexity bound of $O(mn)$, where his $O(mn)$-time algorithm for the `k-RE-MST` subproblem is the bottleneck. Liang [16] improved the complexity of solving the `k-RE-MST` problem to $O(n^2)$, thus achieving the corresponding improvement in Shen's randomized algorithm [19]. We show that our techniques to solve the SLFR problem can be adapted to the `k-RE-MST` problem and solve it in $O(m + n \log n)$ time, thus improving the time complexity of Shen's randomized algorithm [19] for the k-MVE problem from $O(n^2)$ to (near) optimal $O(m + n \log n)$. The decision version of the k-MVE problem is polynomially solvable when k is fixed [19], but for arbitrary k the problem has been shown to be NP-complete by Frederickson and Solis-Oba [7], even when the edge weights are 0 or 1.

1.1 Main Results

Our main results are (near) optimal algorithms for the single link failure recovery (SLFR) problem, a lower bound for the directed 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 the SLFR problem. We present an $O(m + n)$ time algorithm for the case when all the edge weights are the 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, $T_{sort}(n)$ is $O(n \log \log n)$ (Han [11]). The computation of the shortest paths 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.

To exhibit the difference in the difficulty levels of the directed and undirected versions we borrow the lower bound construction of [1, 14] to establish a lower bound of $\Omega(min(n^2, m\sqrt{n}))$ for the arbitrarily weighted directed version of the problem. The construction has been used to establish the same bound for the directed version of the *replacement paths* problem [17, 13, 14, 1]. This lower bound holds in the *path comparison* model for shortest path algorithms.

We show in Section 7 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.

In Section 8 we show that our techniques to solve the SLFR problem can be adapted to the `k-RE-MST` problem and solve it in $O(m+n \log n)$ time, thus improving the time complexity of Shen's randomized algorithm [19] for the k-MVE problem from $O(n^2)$ to (near) optimal $O(m + n \log n)$.

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 cost (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 $E(V_1, V_2)$ represents the set of edges across the cut (V_1, V_2).

A shortest path tree $\mathcal{T}_s$ for a node s is a collection of $n-1$ edges of G, $\{e_1, e_2, \ldots, e_{n-1}\}$, 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 $parent_{\mathcal{T}_s}$ 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.

Finally, $\mathcal{T}_{mst}$ denotes the minimum (cost) spanning tree of the graph, and is a collection of $n - 1$ edges forming a spanning tree with least *total weight* among all spanning trees of the graph.

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

In this section we describe a simple algorithm for the SLFR problem that takes $O(m \log n)$ 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 one edge of the cut $(V_i, V_{s|i})$.

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*

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

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

$$weight(u, v) = d_G(y_i, u) + cost(u, v) + d_G(v, s) \tag{2}$$

Proof: Since G is 2-connected there is at least one path from y_i to s in the graph $G\backslash\{e_i\}$ and all such paths have at least one edge in the cut $E(V_i, V_{s|i})$. We now prove by contradiction that at least one such path has exactly one edge of the cut $(V_i, V_{s|i})$. Suppose that no such path exists. Consider any path π from y_i to s with more than one edge across the cut $E(V_i, V_{s|i})$ (see Figure 1). Let q be the last vertex in the set $V_{s|i}$ visited by π. We define the path π_2 as the path π except that the portion of the path $p_\pi(q, s)$ is replaced by the path from q to s in $\mathcal{T}_s$ which is completely contained within $V_{s|i}$. Since the path in $\mathcal{T}_s$ from q to s is a shortest path in G and does not include edge e_i, it then follows that the weight of path π_2 is less than or equal to that of path π. Clearly, π_2 uses exactly one edge in $E(V_i, V_{s|i})$. A contradiction. This proves the first part of the lemma.

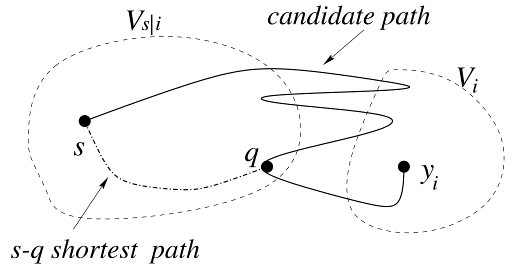

Figure 1: The recovery path to y_i uses exactly one edge across the induced cut

Next, notice that the weight of the candidate path to y_i using the edge $(u, v) \in E(V_i, V_{s|i})$ is exactly equal to $d_G(y_i, u) + cost(u, v) + d_G(v, s)$. This is because the shortest path from y_i to u is completely contained inside V_i and is not affected by the deletion of the edge e_i. Also, since we are dealing with undirected graphs, the shortest path from v to s is of the same weight as the shortest path from s to v which is completely contained inside $V_{s|i}$ and remains unaffected by the deletion of the edge e_i. The minimum among all the candidate paths is the shortest path whose weight is given precisely by the equation (1). $\square$

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. Note that there may be more than one edge that could be an escape edge for each edge e_i. We replace equation (1) with the following equation to compute $\mathcal{A}(e_i)$.

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

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:

$$weight(u,v) = d_G(s,y_i) + d_G(y_i,u) + cost(u,v) + d_G(v,s) = d_G(s,u) + cost(u,v) + d_G(v,s) \tag{4}$$

Now the weight associated with an edge is independent of the cut being considered and we just need 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 $\mathcal{T}_s$. In the first step, we construct $n-1$ heaps, one for each node (except s) 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 incident upon 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 heap.

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 $v_1, v_2, \dots, v_k$. The heap for node v is updated as follows:

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

Now H_v contains all the edges crossing the cut induced by deleting the edge $(parent_{\mathcal{T}_s}(v), v)$. But it may also contain 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 $\mathcal{T}_s$ in the order in which the DFS call to the nodes *end*. 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 $v.min \le dfs(a) < dfs(v)$ and $v.min \le dfs(b) < dfs(v)$. In case e is 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 recovery path from each node to s is computed by our procedure in $O(m \log n)$ time.*

We formally present our algorithm Compute Recovery Paths (CRP) below. Initially one invokes DFS traversal of $\mathcal{T}_s$ where the nodes are labeled in DFS order. At the same time we compute and store in the *min* field of every node, the smallest DFS label among all nodes in the subtree of $\mathcal{T}_s$ rooted at v. We refer to this value as $v.min$. Then one invokes CRP(v) for every child v of s.

```
Procedure CRP (v)
    Construct the heap H_v which initially contains an entry for each non-tree edge incident on it.
    // When v is a leaf in the tree T_s the body of the loop will not be executed //
    for all nodes u such that u is a child of v in T_s do
        CRP(u);
        H_v ← meld(H_v, H_u);
    endfor
    // Now H_v contains all the edges across the induced cut, and when v is not //
    // a leaf in T_s the heap may also contain some invalid ones. //
    // Checking for validity of an edge is a constant time operation as described above. //
    while ( (findMin(H_v)).edge is invalid ) do
        deleteMin(H_v)
    endwhile
    A(parent(v), v) = (findMin(H_v)).edge
    return;
End Procedure CRP
```

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 at least one edge 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 a set $E_{\mathcal{A}}$ 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 3.1 [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 a set $E_{\mathcal{A}}$ 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 [8]. The problem also admits linear time algorithms when edge weights are integers [9]. Improved algorithms are given in [18, 3, 8]. A set $E_{\mathcal{A}}$ contains precisely the edges present in the minimum cost spanning forest (MSF) of $G_{\mathcal{A}}$. The following lemma will establish 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 cost spanning forest of $G_{\mathcal{A}}$. If $\mathcal{A}(e_i)$ is not unique, a minimum cost spanning forest edge offers a recovery path of the same weight.*

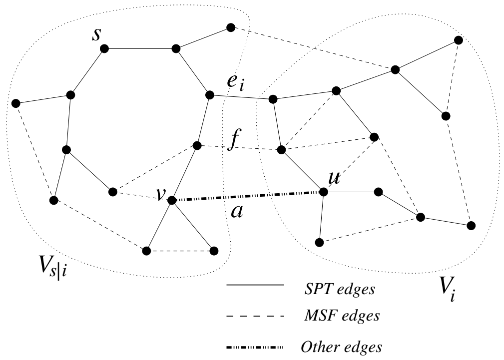

Figure 2: A unique escape edge for e_i has to be an edge in the minimum cost spanning forest of $G_{\mathcal{A}}$.

Proof: Let us assume that for an edge $e_i \in \mathcal{T}_s$, $\mathcal{A}(e_i) = a = (u, v) \in E(V_i, V_{s|i})$ is a *unique* edge not present in the minimum cost spanning forest of $G_{\mathcal{A}}$. Let us investigate the cut $(V_i, V_{s|i})$ in $G(V, E)$. There can be several MSF edges crossing this cut. Since $a = (u, v)$ is in $G_{\mathcal{A}}$, it must be that u and v are in the same connected component in $G_{\mathcal{A}}$. Furthermore, adding a to the MSF forms a cycle in the component of the MSF containing u and v as shown in Figure 2. At least one other edge, say f, of this cycle crosses the cut $(V_i, V_{s|i})$. ¿From Property 3.1 mentioned earlier, $weight(a) \geq weight(f)$ and the recovery path using f in G is at least as good as the one using a. $\square$

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 time complexity of our algorithm is thus $O(m + n \log n)$ which 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 Theorem 3.1.

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`
  Sort the non-tree edges by their weight;
  `for` each non-tree edge $e = (u, v)$ in ascending order `do`
    Let w be the nearest common ancestor of u and v in $\mathcal{T}_s$
    The recovery path 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 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. Here L denotes a priority queue containing the list of edges sorted by increasing order of their *weights*, and supports $deleteMin(\cdot)$ operation in $O(1)$ time.

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.*

Proof: See figure 3. Let us investigate the alternate path for a node y_i lying on $path_{\mathcal{T}_s}(v, w)$. In the graph $G \backslash (x_i, y_i)$ where $x_i = parent_{\mathcal{T}_s}(y_i)$ is the parent of y_i in $\mathcal{T}_s$, we need to find a smallest weighted edge across the cut $(V_i, V_{s|i})$. Note that the path from y_i using e is a valid candidate for the alternate path from y_i since e is an edge across the induced cut. If the alternate path from y_i uses an edge $f \neq e$, then f would have been retrieved by an earlier $deleteMin(L)$ operation and the alternate path from y_i would have already been discovered. Furthermore, if the alternate path from y_i has not been discovered yet, e offers a path at least as cheap as what any other edge across the cut can offer. A similar argument establishes the lemma for the nodes lying on $path_{\mathcal{T}_s}(u, w)$. $\square$

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.

The *nearest common ancestor* problem has been extensively studied. The first linear time algorithm by Harel and Tarjan [12] has been significantly simplified and several linear time algorithms [2] are known for the problem. 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 [21], where α is the functional inverse of Ackermann's function. Although the general problem has a super-linear lower bound [22], a special case of the problem admits linear time algorithm [10]. 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 [10] 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 [10] 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 $\mathcal{T}_s$ using a linear time algorithm [2, 12] to efficiently answer the nearest common ancestor queries.
- Initialize the union-find data-structure of [10].
- 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
    while (find(x) ≠ find(w)) do
      A(parent(x), x) = e;
      union(find(parent(x)), find(x));
      Mark x;
      x = parent(x);
    endwhile
  endfor
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 taken by the sorting, creation of the sorted list and deletion of the smallest element in the list one by one, and the computation of the nearest common ancestor can be shown all to take linear time in the paragraph just before procedure `All-S-L`. It is clear that all the union operations are between a child and a parent in $\mathcal{T}_s$, and the tree $\mathcal{T}_s$ is known ahead of time. Therefore, all the *union-find* operations take $O(n+m)$ time. All the other operations can be shown to take constant time except for the innermost `while` loop which overall takes $O(n)$ time since it is repeated at most once for each edge in the tree $\mathcal{T}_s$. We have thus established Theorem 4.1.

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 Integer Edge Weights SLFR

If the edge weights are integers, linear time algorithms are known for the shortest paths tree [24] and the minimum cost spanning tree [9]. We reduce the number of candidates for the escape edges from $O(m)$ to $O(n)$ using the technique of investigating only the MST edges. After sorting these $O(n)$ 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 [11]. We have thus established the following theorem.

Theorem 5.1 *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.*

6 Directed Graphs

In this section we sketch a super linear (unless $m = \Theta(n^2)$) lower bound of $\Omega(min(n^2, m\sqrt{n}))$ for the directed weighted version of the SLFR problem.

The lower bound construction presented in [1, 14] can be used with a minor modification to establish the claimed result for the SLFR problem. It was used in [1, 14] to prove the same bound for the directed version of the *replacement paths* problem: Given a directed weighted graph G, two specified nodes s and t, and a shortest path $P = \{e_1, e_2, \ldots, e_k\}$ from s to t, compute the shortest path from s to t in each of the k graphs $G \backslash e_i$ for $1 \leq i \leq k$. The bound holds in the *path comparison* model for shortest path algorithms which was introduced in [15] and further explored in [1, 14].

The construction basically reduces an instance of the *n-pairs shortest paths* ($NPSP$)problem to an instance of the SLFR problem in linear time. An $NPSP$ instance has a directed weighted graph H and n specified source-destination pairs (s_j, t_j) in H. One is required to compute the shortest path between each pair, i.e. from s_j to t_j for $1 \leq j \leq n$. For consistency with our problem definition, we need to reverse the directions of all the edges in the construction of [1, 14]. We simply state the main result in this extended abstract. The reader is referred to [1, 14, 15] for the details of the proofs and the model of computation.

Lemma 6.1 *A given instance of an* n-pairs shortest paths *problem can be reduced to an instance of the* SLFR *problem in linear time without changing the asymptotic size of the input graph. Thus, a robust lower bound for the former implies the same bound for the SLFR problem.*

As shown in [1, 14], the $NPSP$ problem has a lower bound of $\Omega(min(n^2, m\sqrt{n}))$ which applies to a subset of path comparison based algorithms. Our lower bound applies to the same class of algorithms to which the lower bound of [1, 14] for the replacement paths problem applies.

7 Alternate Paths Routing for ATM Networks

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.

Let us begin by discussing the inner-working of 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;
    : $q \neq$ l: send (destination s) (message) through q
  endcase
End Routing Protocol

The primary routing table 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 has $parent_{\mathcal{T}_s}(x)$ in the entry for s.

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 in 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 u 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 [20].

While the set of alternate paths generated by the algorithm in Section 3 ensure connectivity, they may introduce loops since the IISP [6] 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 3, 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 [20], storing at most one alternate entry per node does not ensure loop-free routing. E.g. If $\mathcal{A}(w, x_i) = (x_i, a)$ and $\mathcal{A}(s, z) = (y_i, c)$, 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.

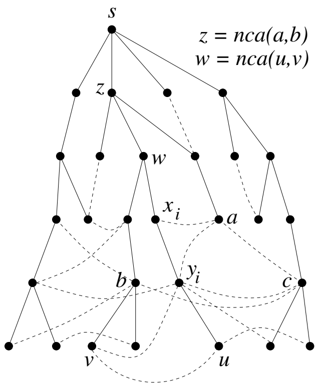

Figure 3: Recovery paths in undirected unweighted graphs

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 $z \in \mathcal{T}_s$ such that $z = child_{\mathcal{T}_s}(r)$, and z is not marked **do**
    $(b, c) = \mathcal{A}(r, z)$ such that $b \in V_z$ (where V_z is the set of vertices in the subtree of $\mathcal{T}_s$ rooted at z)
    while $(b \neq z)$ **do**
      $\mathcal{A}(parent_{\mathcal{T}_s}(b), b) = (b, c)$
      Mark b
      GLAP(b)
      $b = parent_{\mathcal{T}_s}(b)$
    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 7.1 *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.*

8 k-Minimum Replacement Edges in Minimum Cost Spanning Trees

In this section we develop an algorithm for the `k-RE-MST` problem that takes $O(m + n \log n)$ time. We assume that the graph is k edge connected and that k is a constant. The problem was studied by Shen [19] who used it to design a randomized algorithm for the k-MVE problem. Shen's randomized algorithm has a time complexity bound of $O(mn)$, where his $O(mn)$-time algorithm for the `k-RE-MST` subproblem is the bottleneck. Liang improved the complexity of the `k-RE-MST` algorithm to $O(n^2)$, thus achieving the corresponding improvement in Shen's randomized algorithm [19]. The Procedure `CRP` presented in Section 2 can be easily generalized to solve the `k-RE-MST` problem in $O(m+n\log n)$ time, thus improving the time complexity of Shen's randomized algorithm [19] for the *k-most vital arcs in MSTs* problem from $O(n^2)$ to (near) optimal $O(m+n\log n)$.

The idea is to use the algorithm in Section 2 to extract k minimum weight *valid* edges from each heap H_v. Clearly, these k edges are precisely the replacement edges for the edge $(parent(v), v)$. Also, the output of the algorithm is now a set of $n-1$ lists, RE_{e_i} for $1 \leq i \leq n-1$. At the end of the procedure, each list RE_{e_i} contains the k minimum weight replacement edges for the edge e_i. Furthermore, we root $\mathcal{T}_{mst}$ at an arbitrary node $r \in V$, and the weights of the edges are their original weights as defined in the input graph G.

The modification in the Procedure `CRP` is in the `while` loop, which needs to be replaced by the following block:

```
for i = 1 to k, do:
  while ( (findMin(H_v)).edge is invalid ) do
    deleteMin(H_v)
  endwhile
  RE_(parent(v),v).add((deleteMin(H_v)).edge)
endfor
for i = 1 to k, do:
  insert(H_v, RE_(parent(v),v).get(i)).
endfor
```

Note that the second `for` loop is required since an edge in RE_e may appear as one of the edges in RE_f for $f \neq e$. Now we analyze the complexity of this modified Procedure `CRP`. The `while` loop performs at most $O(m)$ $deleteMin(\cdot)$ operations over the entire execution of the algorithm, thus contributing an $O(m\log n)$ term. The first and second `for` loops in the block above, perform additional k $deleteMin(\cdot)$ and $insert(\cdot)$ operations respectively, per heap H_v. The $add(\cdot)$ and $get(\cdot)$ list operations are constant-time operations. The remaining steps of the algorithm are same as for the $SLFR$ problem. Thus, the total time complexity of this procedure is $O(m\log n + kn\log n) = O(m\log n)$ (since k is *fixed*).

In a preprocessing step, we reduce the number of edges of $G(V,E)$ from $O(m)$ to $(k+1)(n-1)$ which is $O(n)$ using the following lemma established by Liang [16].

Lemma 8.1 [16] *If $\mathcal{T}_1$ is the MST/MSF of $G(V,E)$, and $\mathcal{T}_i$ is the MST/MSF of $G_i = G(V, E \setminus \cup_{j=1}^{i-1}\mathcal{T}_j)$, for $i > 1$, then $U_{k+1} = \cup_{j=1}^{k+1}\mathcal{T}_j$ contains the k-minimum weight replacement edges for every edge $e \in \mathcal{T}_1$.*

The set U_{k+1} can be easily constructed in $O((k+1)(m+n\log n)) = O(m+n\log n)$ time by invoking a standard MST algorithm $k+1$ times. Now, the above modified `CRP` procedure takes only $O(m+n\log n)$ time.

Theorem 8.1 *Given a k-edge connected graph, where k is a constant, our procedure defined above takes $O(m+n\log n)$ time to solve the k-minimum weight replacement edges for every edge $e \in \mathcal{T}_s$,*

9 Concluding Remarks

In this paper we have presented near optimal algorithms for the undirected version of the SLFR problem and a lower bound for the directed version. In Section 8, we modified the basic algorithm of Section 2 to derive a (near) optimal algorithm for the `k-RE-MST` problem, which finds application in Shen's randomized algorithm for the `k-MVE` problem on MSTs.

One obvious open question is to bridge the gap between the lower bound and the naive upper bound for the directed version. The directed version is especially interesting since an $O(f(m,n))$ time algorithm for it implies an $O(f(m+k, n+k))$ time algorithm for the k-pairs shortest paths problem for $1 \le k \le n^2$.

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.

A slight modification of our algorithm also enables us to efficiently compute the recovery paths when a *node* of the (undirected) network fails.A narrow gap also remains to be filled for the undirected version when edge weights are arbitrary integers.

When there is more than one possible destination in the network, one needs to apply the algorithms in this paper for each of the destinations.

References

[1] A. Bhosle. On the difficulty of some shortest paths problems, Master's thesis, University of California, Santa Barbara, http://www.cs.ucsb.edu/~bhosle/publications/msthesis.ps. 2002.

[2] 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 *30th ACM STOC*, pages 279-288. ACM Press, 1998.

[3] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *JACM*, 47:1028-1047, 2000.

[4] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.

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

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

[7] G. N. Frederickson and R. Solis-Oba. Increasing the weight of minimum spanning trees. In *Proceedings of the 7th ACM/SIAM Symposium on Discrete Algorithms*, pages 539–546, 1993.

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

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

[10] 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.

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

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

[13] J. Hershberger and S. Suri. Vickrey prices and shortest paths: What is an edge worth? In *42^{nd} IEEE FOCS*, pages 252-259, 2001.

[14] J. Hershberger, S. Suri, and A. M. Bhosle. On the difficulty of some shortest path problems. In *20^{th} STACS*, pages 343–354. Springer-Verlag, 2003.

[15] D. R. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. In *32^{nd}IEEE FOCS*, pages 560-568, 1991.

[16] W. Liang. Finding the k most vital edges with respect to minimum spanning trees for fixed k. *Discrete Applied Mathematics*, 113:319–327, 2001.

[17] K. Malik, A. K. Mittal, and S. K. Gupta. The k most vital arcs in the shortest path problem. In *Oper. Res. Letters*, pages 8:223-227, 1989.

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

[19] Hong Shen. Finding the k most vital edges with respect to minimum spanning tree. *Acta Informatica*, 36(5):405–424, 1999.

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

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

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

[23] R. E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path problems. *Inform. Proc. Lett.*, 14:30–33, 1982.

[24] Mikkel Thorup. Undirected single source shortest path in linear time. In *38^{th} IEEE FOCS*, pages 12–21, 1997.