Efficient Algorithms and Routing Protocols for Handling Transient Single Node Failures

Amit M. Bhosle* and Teofilo F. Gonzalez

Department of Computer Science
University of California
Santa Barbara, CA 93106
{bhosle,teo}@cs.ucsb.edu

Abstract

Single *node failures represent more than 85% of all node failures in the today's large communication networks such as the Internet [10]. Also, these node failures are usually* transient. *Consequently, having the routing paths globally recomputed does not pay off since the failed nodes recover fairly quickly, and the recomputed routing paths need to be discarded. Instead, we develop algorithms and protocols for dealing with such transient single node failures by suppressing the failure (instead of advertising it across the network), and route messages to the destination via alternate paths that do not use the failed node. This philosophy was also adopted in Ref. [14] where the authors address the same issues. We develop algorithms which are faster than those given in Ref. [14] by an order of magnitude, while our paths are equally good. We show via simulation results that our paths are usually within 9-12% of the optimal for randomly generated graph with 100-1000 nodes.*

KEY WORDS: *Network Protocols, Fault Tolerant Systems, Node Failure Recovery, Transient Node Failures, Alternate Path Routing.*

1 Introduction

Computer networks are normally represented by edge weighted graphs. The vertices represent computers, the edges represent the direct communication links between pairs of computers, and the weight of an edge represents the cost (e.g. time) required to transmit a message (of some given length) through the link. The links are bi-directional. Given a computer network represented by an edge weighted graph $G = (V, E)$, the problem is to find the best route (under normal operation load) to transmit a message between every pair of vertices. The number of vertices ($|V|$) is n and the number of edges ($|E|$) is m. The shortest paths tree of a node s, $\mathcal{T}_s$, specifies the fastest way of

*Currently at Amazon.com, 1200 12^{th} Ave. S., Seattle, WA - 98144

transmitting a message to node s originating at any given node in the graph. Of course, this holds as long as messages can be transmitted at the specified costs. When the system carries heavy traffic on some links these routes might not be the best routes, but under normal operation the routes are the fastest. It is well known that the all pairs shortest path problem, finding a shortest path between every pair of nodes, can be computed in polynomial time. In this paper we consider the case when the nodes in the network may be susceptible to transient faults. These are sporadic faults of at most one node[1] at a time that last for a relatively short period of time. This type of situation has been studied in the past [14] because it represents most of the node failures occurring in networks. *Single* node failures represent more than 85% of all node failures[10]. Also, these node failures are usually *transient*, with 46% lasting less than a minute, and 86% lasting less than 10 minutes[10]. Because nodes fail for relative short periods of time, propagating information about the failure throughout the network is not recommended. The reason for this is that it takes time for the information about the failure to be communicated to all nodes and it takes time for the nodes to recompute the shortest paths in order to re-adapt to the new network environment. Then, when the failing node recovers, a new messages disseminating this information needs to be sent to inform the nodes to roll back to the previous state. This process also consumes resources. Therefore, propagation of failures is best suited for the case when nodes fail for long periods of time. This is not the scenario which characterizes current computer networks, and it is not considered in this paper.

In this paper we consider the case where the network is *biconnected* (*2-node-connected*), meaning that the deletion of a single node does not disconnect the network. Biconnectivity ensures that there is at least one path between every pair of nodes even in the event that a node fails (provided the failed node is not the origin or destination of a path). A ring network is an example of a biconnected network, but it is not necessary for a network to have a ring formed by all of its nodes in order to be biconnected. Testing whether or not a network is biconnected can be performed in linear time with respect to the number of nodes and links in a network. The algorithm is based on depth-first search [13].

Based on our previous assumptions about failures, 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 node (other than s) that failed. In the latter case, we need to use a recovery path to s from that point. Since we assume single node faults and the graph is biconnected, such a path always exists. We call this problem of finding the recovery paths the *Single Node Failure Recovery (SNFR)* problem. It is important to recognize that the recovery path depends heavily on the protocol being deployed in the system. In this paper we discuss several routing protocols.

1.1 Related Work

The SNFR problem was studied by Zhang, et. al. in [14], wherein the authors have presented protocols based on local re-routing for dealing with transient single node failures. The alternate paths are precomputed at network setup time. Information about single node failures in the network is suppressed, and instead, messages are rerouted using these precomputed alternate paths. They demonstrate via simulations that the recovery paths computed by their algorithm are usually within 15% of the theoretically optimal recovery paths.

A closely related problem to our problem is the Single *Link* Failure Recovery problem. In the SLFR problem there is only one *link* failure, the failure is transient, and information about the

[1]The nodes are *single*- or *multi*-processor computers

failure is not propagated throughout the network. 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. The link version of the problem is relatively easier than the node version: the former has been shown to admin an optimal algorithm with a running time of $O(m + n \log n)$ time in [1]. Also, the version of the SLFR problem is to find an *optimal* (shortest) recovery paths.

One of the main applications of the SLFR problem is the *alternate path routing* (APR) problem for communications networks. This problem arises when using a special protocol (see Bhosle and Gonzalez [1]). This protocol 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. Slosiar and Latin [12] had studied this problem and presented an $O(n^3)$ time algorithm. Bhosle and Gonzalez [1] developed an $O(m + n \log n)$ time algorithm for APR problem.

1.2 Organization of the paper

We formally define the SNFR problem in Section 1.3. The notation that we adopt in the rest of the paper is defined in Section 1.4, followed by our main results in Section 1.5. We describe our central algorithm in detail in Section 2, and discuss the protocol based on the SNFR algorithm in Section 3. Simulation results and comparison with the results of [14] are presented in Section 4, and we conclude the paper with a short discussion in Section 5.

1.3 Problem Definition

The Single Node Failure Recovery problem, is defined as follows:

SNFR: Given a biconnected undirected edge weighted graph $G = (V, E)$, and the shortest paths tree $\mathcal{T}_s(G)$ of a node s in G where $\mathcal{C}_x = \{x_1, x_2, \ldots x_{k_x}\}$ denotes the set of *children* of the node x in $\mathcal{T}_s$, for each node $x \in V$ and $x \neq s$, find a path from $x_i \in \mathcal{C}_x$ to s in the graph $G = (V \backslash \{x\}, E \backslash E_x)$, where E_x is the set of edges adjacent to vertex x.

In other words, for each node x in the graph, we are interested in finding alternate paths from each of its children to the source node s when the node x *fails*. Note that node x cannot be node s.

1.4 Preliminaries

Our communication network is modeled by an edge-weighted biconnected undirected graph $G = (V, E)$, with $n = |V|$ and $m = |E|$. Each edge $e \in E$ has an associated cost (weight), denoted by $cost(e)$, which is a non-negative real number. We use $p_G(s, t)$ to denote a shortest path between s and t in graph G and $d_G(s, t)$ to denote its cost (weight).

A shortest path tree $\mathcal{T}_s$ for a node s is a collection of $n - 1$ edges $\{e_1, e_2, \ldots, e_{n-1}\}$ of G which form a spanning tree of G such that the path from node v to s in $\mathcal{T}_s$ is a shortest path from v to s in G. We say that $\mathcal{T}_s$ is rooted at node s. With respect to this root we define the set of nodes that are the children of each node x as follows. In $\mathcal{T}_s$ we say that every node y that is adjacent to x such

that x is on the path in $\mathcal{T}_s$ from y to s, is a child of x. For each node x in the shortest paths tree, k_x denotes the number of *children* of x in the tree, and $\mathcal{C}_x = \{x_1, x_2, \ldots x_{k_x}\}$ denotes this set of children of the node x. Also, x is said to be the *parent* of each $x_i \in \mathcal{C}_x$ in the tree $\mathcal{T}_s$.

$V_x(\mathcal{T})$ denotes the set of nodes in the subtree of x in the tree $\mathcal{T}$ and $E_x \subset E$ denotes the set of all edges incident on the node x in the graph G. We use $nextHop(x, y)$ to denote the next node from x on the shortest path tree from x to y. Note that by definition, $nextHop(x, y)$ is the parent of x in $\mathcal{T}_y$.

1.5 Main Results

We present an efficient[2] algorithm for the SNFR problem that has a running time of $O(m \log n)$. We further develop protocols based on this algorithm for recovering from single node *transient* failures in communication networks.

The recovery paths computed by our algorithm are not necessarily the shortest recovery paths. However, we demonstrate via simulation results that they are very close to the optimal paths. The test data consists of randomly generated graphs with an average *stretch* factor[3] of the paths within 4-12% of the optimal in graphs with 100-1000 nodes.

Finally, we compare our results with those of [14] wherein the authors have also studied the same problem and presented protocols based on local rerouting for dealing with transient single node failures. One important difference between the algorithm of [14] and our's is that their's performs a lot of recomputations, while our's reuses information computed in previous steps of the algorithm. Consequently, our algorithm is faster by an order of magnitude than those in [14], and as shown by our simulation results, our recovery paths are usually comparable, and sometimes better.

2 Algorithm for Single Node Failure Recovery

In this section we describe an algorithm for the SNFR problem defined in Section 1.3 and later develop protocols for recovering from transient single node failures based on the SNFR algorithm.

A naive algorithm for the SNFR problem is based on recomputation: for each node $v \in G(V)$ and $v \neq s$, compute the shortest paths tree of s in the graph $G(V \backslash v, E \backslash E_v)$. Of interest are the paths from s to each of the nodes $v_i \in \mathcal{C}_v$. This naive algorithm invokes a shortest paths algorithm $n-1$ times, and thus takes $O(mn + n^2 \log n)$ time when it uses the Fibonacci heap[4] implementation of Dijkstra's shortest paths algorithm[3]. While these paths are *optimal* recovery paths for recovering from the single node failure, their *structure* can be much different from each other, and from the original shortest paths (in absence of any failures) - to the extent that routing messages along these paths may involve recomputing large parts of the primary routing tables at the nodes through which these paths pass. Also, it may not be easy to switch between the alternate paths and primary paths as nodes fail or recover in the network. As we shall see later, the recovery paths computed by our algorithm have a well defined structure, and they overlap with the paths in the original shortest paths tree ($\mathcal{T}_s$) to an extent that storing the information of a single edge at

[2]The primary routing tables can be computed using the Fibonacci heaps[4] based implementation of Dijkstra's shortest paths algorithm[3] in $O(m + n \log n)$ time

[3]The stretch is defined in Ref. [14], as the ratio of the lengths of the recovery paths computed by an algorithm for the SNFR problem to the lengths of the optimal recovery paths

each node provides sufficient information to infer the entire recovery path. Moreover, our approach makes it easy to switch between alternate and primary paths as the nodes of the network fail and recover.

2.1 Basic Principles and Observations

We start by describing some basic observations about the characteristics of the recovery paths. We also categorize the graph edges according to their *role* in providing recovery paths for a node when its parent fails.

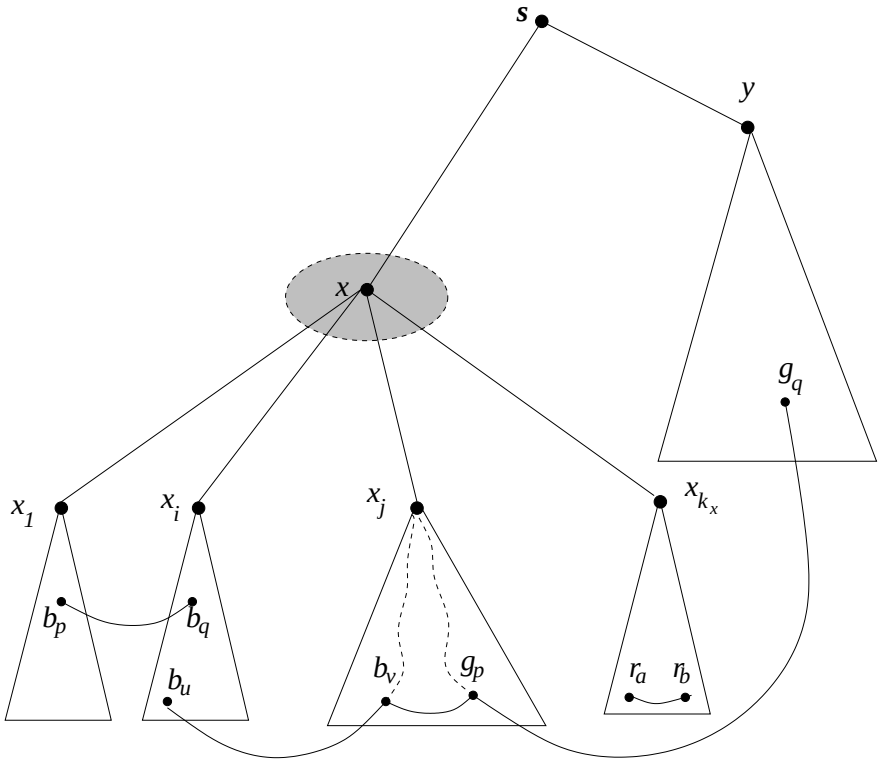


Figure 1. Recovery paths for recovering from the failure of x.

Figure 1 illustrates a scenario of a single node failure. In this case, the node x has failed, and we need to find recovery paths to s from each $x_i \in \mathcal{C}_x$. When a node fails, the shortest paths tree of s, $\mathcal{T}_s$, gets split into $k_x + 1$ components - one containing the source node s and each of the remaining ones contain one subtree of a child $x_i \in \mathcal{C}_x$.

Notice that the edge $\{g_p, g_q\}$ (Figure 1), which has one end point in the subtree of x_j, and the other outside the subtree of x provides a candidate recovery path for the node x_j. The complete path is of the form $p_G(x_j, g_p) \rightsquigarrow \{g_p, g_q\} \rightsquigarrow p_G(g_q, s)$. Since g_q is outside the subtree of x, the path $p_G(g_q, s)$ is not affected by the failure of x. Edges of this type (from a node in the subtree of $x_i \in \mathcal{C}_x$ to a node outside the subtree of x) can be used by $x_i \in \mathcal{C}_x$ to *escape* the failure of node x. These edges are called *green* edges. For example, edge $\{g_p, g_q\}$ is a green edge.

Next, consider the edge $\{b_u, b_v\}$ (Figure 1) between a node in the subtree of x_i and a node in the subtree of x_j. Although there is no green edge with an end point in the subtree of x_i, the edges $\{g_p, g_q\}$ and $\{b_u, b_v\}$ together offer a candidate recovery path that can be used by x_i to recover

from the failure of x. Part of this path connects x_i to x_j ($p_G(x_i, b_u) \rightsquigarrow \{b_u, b_v\} \rightsquigarrow p_G(b_v, x_j)$), after which it uses the recovery path of x_j (via x_j's green edge, $\{g_p, g_q\}$). Edges of this type (from a node in the subtree of x_i to a node in the subtree of x_j for some $i \neq j$) are called *blue* edges. Another example of a blue edge is edge $\{b_p, b_q\}$ which can be used the node x_1 to recover from the failure of x.

Note that edges like $\{r_a, r_b\}$ and $\{b_v, g_p\}$ (Figure 1) with both end points within the subtree of the same child of x do not help any of the nodes in $\mathcal{C}_x$ to find a recovery path from the failure of node x. We do not consider such edges in the computation of recovery paths, even though they may provide a shorter recovery path for some nodes (e.g. $\{b_v, g_p\}$ may offer a shorter recovery path to x_i). The reason for this is that routing protocols would need to be very complex in order to use this information. As we describe later in the paper, we carefully organize the *green* and *blue* edges in a way that allows us to retain only the useful edges and eliminate useless (red) ones efficiently.

We now describe the construction of a new graph G_x which will be used to compute recovery paths for the elements of $\mathcal{C}_x$ when the node x fails. As we later show, a single source shortest paths computation on this graph suffices to compute the recovery paths for all $x_i \in \mathcal{C}_x$.

The graph G_x has $k_x + 1$ nodes, where $k_x = |\mathcal{C}_x|$. A special node, s_x, represents the source node s in the original graph $G = (V, E)$. Apart from s_x, we have one node, denoted by y_i, for each $x_i \in \mathcal{C}_x$. We add all the *green* and *blue* edges defined earlier to the graph G_x as follows. A green edge with an end point in the subtree of x_i (by definition, green edges have the other end point outside the subtree of x) translates to an edge between s_x and y_i. A blue edge with an end point in the subtree of x_i and the other in the subtree of x_j translates to an edge between nodes y_i and y_j. However, the weight of each edge added to G_x is not the same as the weight of the green or blue edge in $G = (V, E)$ used to define it. The weights for these edges is specified below.

Note that the candidate recovery path of x_j that uses the green edge $g = \{g_p, g_q\}$ has total cost equal to:

$$greenWeight(g) = d_G(x_j, g_p) + cost(g_p, g_q) + d_G(g_q, s) \tag{1}$$

This weight *can*[4] be assigned to the edge corresponding to the green edge $\{g_p, g_q\}$ that is added in G_x between y_j and s_x. If there are multiple green edges with an end point in V_{x_j}, the subtree of x_j, we choose the one which offers the shortest recovery path for y_j, and ignore the rest (with ties being broken arbitrarily).

As discussed earlier, a blue edge provides a path connecting two siblings of x, say x_i and x_j. Once the path reaches x_j, the remaining part of the recovery path of x_i coincides with that of x_j. If $\{b_u, b_v\}$ is the blue edge connecting the subtrees of x_i and x_j (the cheapest one corresponding to the edge $\{y_i, y_j\}$), the length of the subpath from x_i to x_j is:

$$blueWeight(b) = d_G(x_i, b_u) + cost(b_u, b_v) + d_G(b_v, x_j) \tag{2}$$

We assign this weight to the edge corresponding to the blue edge $\{b_u, b_v\}$ that is added in G_x between y_i and y_j.

[4]Since the weight given by equation (1) for an edge depends on the node x_j whose recovery path is being computed, using these weights directly involves the expensive task of updating these values as we look at different nodes. Later on we define and use a slightly different weight function for the edges which is independent of the node whose recovery path is being computed.

The construction of our graph G_x is now complete. As we discuss later, computing the shortest paths tree of s_x in G_x provides enough information to compute the recovery paths for all nodes $x_i \in \mathcal{C}_x$ when x fails.

2.2 Description of the Algorithm and its Analysis

We now incorporate the basic observations described earlier into a formal algorithm for the SNFR problem. Then we analyze the complexity of our algorithm and show that it has a nearly optimal running time of $O(m \log n)$.

Our algorithm is a *depth-first* recursive algorithm over $\mathcal{T}_s$. We maintain the following information at each node x:

- Green Edges: The set of green edges in $G = (V, E)$ that offer a recovery path for x to escape the failure of its parent.
- Blue Edges: A set of edges $\{p, q\}$ in $G = (V, E)$ such that x is the nearest-common-ancestor of p and q with respect to the tree $\mathcal{T}_s$.

Green Edges

The set of green edges for node x is maintained in a *min heap* (*priority queue*) data structure, which is denoted by $\mathcal{H}_x$. The heap elements are tuples of the form $< e, greenWeight(e) >$ where e is a green edge, and $greenWeight(\cdot)$ defines its priority as an element of the heap. Since the weight of the path using a green edge is given by the equation (1), the $greenWeight(\cdot)$ function can be defined as in equation (1). However, the expression in this equation depends on the node x_i whose recovery path is being computed. Thus, using this weight would imply additional computations to update the weight of the edges as they move from the $\mathcal{H}$ heaps of a node to those of their parents (and then ancestors'). We get around this by further analyzing the use of green edges in the recovery paths for a node x. Once we have a minimum weight edge in $\mathcal{H}_x$, we do not need to consider other elements of $\mathcal{H}_x$ for computing the recovery path for x. We use the following $greenWeight(\cdot)$ function as the priority for the heap element corresponding to the edge $g = \{p, q\}$:

$$
\begin{aligned}
&greenWeight(g) \\
&\qquad = d_G(s, x) + d_G(x, p) + cost(p, q) + d_G(q, s) \\
&\qquad = d_G(s, p) + cost(p, q) + d_G(q, s) \qquad (3)
\end{aligned}
$$

I.e., when the edges are part of $\mathcal{H}_x$ we have added the term $d_G(s, x)$ to the weight function defined in equation (1). Interestingly, the relative ordering of the edges in $\mathcal{H}_x$ does not change when we change our $greenWeight(\cdot)$ function from equation (1) to (3). However, equation (3) is better behaved in that it does not depend on the node x whose heap ($\mathcal{H}_x$) contains the entry corresponding to the green edge. In other words the weight of the green edge is invariant over all the nodes where it is a green edge.

Since an edge can serve as a green edge for $O(n)$ nodes, storing them explicitly at all nodes would impose high space (and time) requirements. Instead of storing the green edges explicitly, we compute this set dynamically when required. Dynamically building this set for a node x involves merging the $\mathcal{H}$ heaps of the children x_i of x. However, an edge which is a green edge for a node

x_i may not remain a green edge for its parent, x. We get around this limitation by using a quick constant-time check which tells us whether the edge is a green edge for x or not. This quick check is only performed when the edge is the top entry (minimum *key*) in the heap. If it is no longer a green edge, it will be deleted and the next edge is considered.

We use the dfs-labeling technique described in [1] to determine whether an edge is a green edge for a node or not. The dfs-labeling assigns a *start* and *end* dfs label to each node as a depth-first-search traversal of $\mathcal{T}_s$ starts or ends at the node. Let $dfsStart(\cdot)$ and $dfsEnd(\cdot)$ denote these labels. It follows from the property of these dfs labels that a node u belongs to the subtree V_v of a node v iff $dfsStart(v) < dfsStart(u) < dfsEnd(u) < dfsEnd(v)$.

Given the above property of the dfs labels, an edge $e = \{p, q\}$ is a green edge for a node u iff exactly one of p and q belongs to the subtree of u, while the other lies outside the subtree of the *parent* of u in $\mathcal{T}_s$.

Blue Edges

Note that the edges $\{b_u, b_v\}$ and $\{b_p, b_q\}$ in Figure 1 are *red* for the node x itself (since both their end points are within V_x, the subtree of x), but *blue* for the children of x. We store the set of blue edges for a node at the node's parent in a set denoted by $\mathcal{B}$. This way, an edge belongs to the $\mathcal{B}$ set of exactly one node (which is the nearest-common-ancestor of its two end points).

The *nearest common ancestor* problem has been extensively studied. The first linear time algorithm by Harel and Tarjan[5] has been simplified and several linear time algorithms (e.g. [2]) are known for the problem. Using these algorithms, after a linear time preprocessing, one can find the nearest common ancestor of any two specified nodes in a given tree in constant time. Using this data structure, we can build the $\mathcal{B}$ set for all nodes in $O(m)$ total time.

The weights of these blue edges added to G_x are as given by the equation (2). Consider $b = \{p, q\} \in \mathcal{B}_x$, which serves as a blue edge for the children of x. We add an edge corresponding to b in G_x between the nodes p_x and q_x where p_x and q_x denote respectively the children of x whose subtree contains the actual end points p and q of b. We now outline the process of determining these two children p_x and q_x of x. We suggest two simple options for this part:

(a) All pairs shortest paths: If we have the all pair shortest paths information for all nodes of G, then the next hop from x to p will be the child of x whose subtree contains p. That is, p_x=$nextHop(x, p)$ and similarly, q_x=$nextHop(x, q)$. These are constant-time lookups once the all pair shortest paths primary lookup tables are available.

(b) *dfs* labels: As mentioned earlier, the $dfsStart(\cdot)$ and $dfsEnd(\cdot)$ labels can be used to determine in constant time whether a node u lies in the subtree of a node v. Also note that the $dfsStart(\cdot)$ and $dfsEnd(\cdot)$ values of each node $x_i \in \mathcal{C}_x$ partition the $(dfsStart(x), dfsEnd(x))$ range in disjoint segments of $(dfsStart(x_i), dfsEnd(x_i))$. Once we sort elements $x_i \in \mathcal{C}_x$ according to the $dfsStart(x_i)$ values, p_x can be found using binary search for the value of $dfsStart(p)$ in $O(\log n)$ time.

If there are multiple edges in $\mathcal{B}_x$ which translate to an edge between the same two nodes of G_x, we retain the one with the cheapest weight, and discard the rest.

Computing the Recovery Paths

Initially $\mathcal{H}_x$ contains an entry for each edge of x which serves as a green edge for it (i.e. an edge of x whose other end point does not lie in the subtree of the parent of x). The elements of $\mathcal{B}_x$ are tuples $< e, blueWeight(e) >$, maintained as a linked list. The $blueWeight(\cdot)$ function is given by the equation (2).

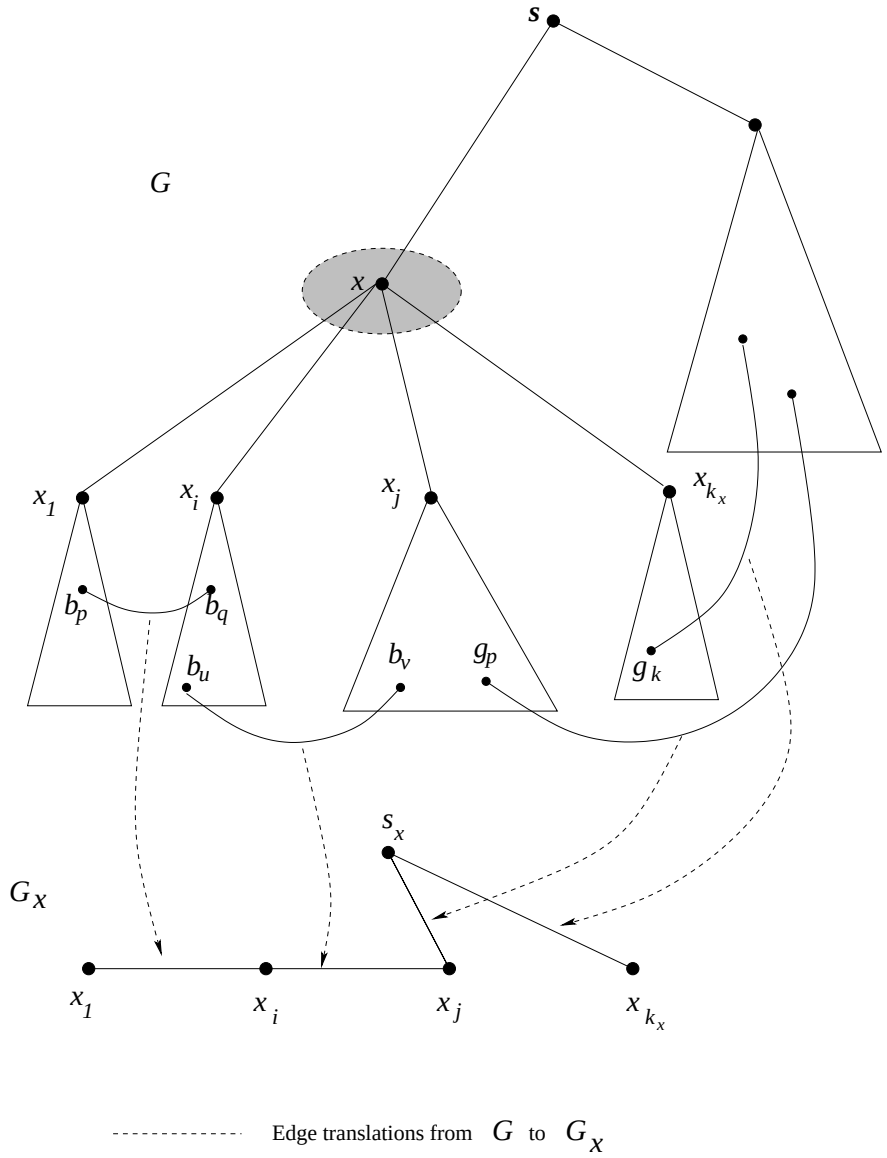


Figure 2. Recovery paths for recovering from the failure of x: Constructing G_x

Figure 2 illustrates an arbitrary step in the algorithm where the the recovery paths from the node $x_i \in \mathcal{C}_x$ to the node s need to be computed when the node x has failed. The recovery paths from the children of x_i to s for recovering from the failure of x_i have been computed in the previous recursions of the algorithm.

For computing the recovery paths for the elements of $\mathcal{C}_x$, we build the $(k_x + 1)$-node graph G_x as described in Section 2.1. Since the elements in $\mathcal{H}_{x_i}$ have the weights given by equation (3), we

subtract the term $d_G(s, x_i)$ from the weight of the edge retrieved from $\mathcal{H}_{x_i}$. This way, we use the correct weight for the minimum weight green edge, as given by equation (1).

As we mentioned before, $\mathcal{H}_{x_i}$ is built by merging together the $\mathcal{H}$ heaps of the nodes in $\mathcal{C}_{x_i}$, the set of children on x_i. Consequently, all the elements in $\mathcal{H}_{x_i}$ may not be green edges for x_i. Using the dfs labeling scheme, we can quickly determine whether the edge retrieved by $findMin(\mathcal{H}_{x_i})$ is a valid green edge for x_i or not. If not, we remove the entry corresponding to the edge from $\mathcal{H}_{x_i}$ via a $deleteMin(\mathcal{H}_{x_i})$ operation. Note that since the deleted edge cannot serve as a green edge for x_i, it cannot serve as one for any of the ancestors of x_i, and it doesn't need to be added back to the $\mathcal{H}_x$ heap for any x. We continue deleting the minimum weight edges from $\mathcal{H}_{x_i}$ till either $\mathcal{H}_{x_i}$ becomes empty or we find a green edge valid for x_i to escape x's failure, in which case we add it to G_x.

After adding the green edges to G_x, we add the blue edges from $\mathcal{B}_x$ to G_x.

Finally, we compute the shortest paths tree of the node s_x in the graph G_x using a standard shortest paths algorithm, Dijkstra's algorithm[3]. The *escape edge* for the node x_i is stored as the *parent edge* of x_i in $\mathcal{T}_{s_x}$, the shortest paths tree of s_x in G_x.

Since the communication graph is assumed to be *bi-connected*, there exists a path from each node $x_i \in \mathcal{C}_x$ to s_x, provided that the failing node is not note s. Also, since this is a tree, we are guaranteed to have acyclic paths from each x_i to s_x.

We formally present our algorithm Compute Recovery Paths (CRP) below. The algorithm is initially invoked on the node s.

Precomputation

Compute the $dfsStart(\cdot)$ and $dfsEnd(\cdot)$ labels for the nodes based on the depth-first-search of $\mathcal{T}_s$.
Initialize the nearest-common-ancestor data-structure of [5] or [2], and assign each graph edge $\{u, v\}$ to the Blue Edges set $\mathcal{B}$ of the node $nca(u, v)$.
For each node $v \in G(V)$ and $v \neq s$, initialize the heap $\mathcal{H}_v$ which initially contains an entry for each green edge incident on it. The weights of the green edges are as defined in equation (3).

Procedure CRP (x)

// Computes the recovery paths for $x_i \in \mathcal{C}_x$ when x fails
`if` x is a `leaf` in $\mathcal{T}_s$ `return;`

// Recurse on the children of x
`for` each node $x_i \in \mathcal{C}_x$ `do`
  CRP(x_i);
`end-for`

`if` $x = s$ `return;` // Everything done

// Build the subgraph $G_x(V_x, E_x)$ as defined in Section 2.1
$V_x = \{s_x\}$ // s_x represents the source node s in G_x
`for` each node $x_i \in \mathcal{C}_x$ `do`
  V_x`.add`(x_i)

// Find the cheapest green edge from $\mathcal{H}_{x_i}$, but delete all the `red` edges encountered
`while` $\mathcal{H}_{x_i}$`.findMin()` is red `do`
$\mathcal{H}_{x_i}$`.deleteMin()`
`end-while`

// If there's another edge in $\mathcal{H}_{x_i}$, use it
`if` ($\mathcal{H}_{x_i}$`.size()`> 0) `do`
$e = \mathcal{H}_{x_i}$`.findMin()`
$w = greenWeight(e) - d_G(s, x_i)$
E_x`.add(new Edge(`x_i`,` s_x`,` w`))`
`end-if`
`end-for`

// Add the blue edges from $\mathcal{B}_x$ to E_x
`for` each edge $b = \{p, q\} \in \mathcal{B}_x$ `do`
$p_x \leftarrow$ child of x whose subtree contains p
$q_x \leftarrow$ child of x whose subtree contains q
$w = blueWeight(e)$
E_x`.add(new Edge(`p_x`,` q_x`,` w`))`
`end-for`

Compute the shortest paths tree $\mathcal{T}_{s_x}$ of s_x in $G_x(V_x, E_x)$

`for` each node $x_i \in \mathcal{C}_x$ `do`
$\mathcal{P}_{x_i}$ = `parent` node of x_i in $\mathcal{T}_{s_x}$
x_i`.escapeEdge` $= \{x_i, \mathcal{P}_{x_i}\}$
`end-for`

`for` each node $x_i \in \mathcal{C}_x$ `do`
$H_x \leftarrow meld(H_x, H_{x_i})$;
`end-for`

End Procedure CRP

The analysis of the above algorithm is straightforward and its time complexity is dominated by the heap operations involved on the heaps $\mathcal{H}$. Using amortized heaps (e.g. Fredman and Tarjan's F-Heaps[4]) we can perform the operations $findMin$ and $meld$ in amortized constant time, while $deleteMin$ requires $O(\log n)$ amortized time.

Computing the shortest paths tree of s_x in G_x takes time $O(m_x + n_x \log n_x)$ where m_x is the number of edges in G_x and n_x is the number of nodes. From the construction of G_x, it follows that $n_x = k_x + 1$. There are at most k_x green edges added to G_x (at most one green edge is added between s_x and the node corresponding to $x_i \in \mathcal{C}_x$), and as discussed earlier, an edge of $G(V, E)$ belongs to exactly one set $\mathcal{B}_x$. I.e.

$$m_x \leq |\mathcal{B}_x| + k_x$$

$$\sum |\mathcal{B}_x| = O(m)$$

Over the entire course of the algorithm, an edge of $G = (V, E)$ is deleted from any $\mathcal{H}$ heap at most once. The overall time complexity of the algorithm can be shown to be $O(m \log n)$ where $m = |E|$ and $n = |V|$. 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 recovery path from each node x_i to s to escape from the failure of the parent of x is computed by our procedure in $O(m \log n)$ time.*

3 Single Node Failure Recovery Protocol

We now describe a protocol based on the *SNFR* algorithm that is designed to deal with transient single node failures in communication networks. As mentioned earlier, the information about a node's failure is suppressed, and not advertised across the network. Instead, the nodes that detect the failure re-route the messages via alternate paths which do not use the failed node.

For an illustration of how the protocol works, consider the network G in Figure 2. If x_i notices that x has failed, it adds information in the message header about $\{b_u, b_v\}$ as the escape edge to use, and reroutes the message to b_u. b_u clears the header information, and sends the message to b_v via the edge (b_u, b_v), after which it follows the *regular* path to s. If x has not recovered when the message reaches x_j, x_j reroutes with message to g_p with $\{g_p, g_q\}$ as the escape edge to use. This continues till the message reaches a node outside the subtree of x, or till x recovers.

Note that when routing a message to a node s, if a node x needs to forward the message to another node y, the node y is the *parent* of x in the shortest paths tree $\mathcal{T}_s$ of s. The SNFR algorithm computes the recovery path from x to s which does not use the failed node y. In case a node has failed, the protocol re-routes the messages along these alternate paths that have been computed by the SNFR algorithm.

In one version of our protocol, the node x that discovers the failure of y adds information about the escape edge to use in the message header. The escape edge is same as the one identified for the node x to use when its parent (y, in this example) has failed. We take advantage of the fact that TCP headers are not of fixed size, and other header fields (e.g. data offset) indicate where the actual message data begins. For our purpose, we need an additional header space for two node identifiers (e.g. IP addresses, and the port numbers) which define the two end points of the escape edge. It is important to note that this extra space is required only when the messages are being re-routed as part of the single node failure recovery. In absence of failures, we do not need to modify the message headers.

For this discussion, we assume the message packet has the following structure:

$$message(s|d_1|d_2|data)$$

where s denotes the final destination d_1, and d_2 denotes the intermediate destinations, and $data$ denotes the actual message data in the packet.

Note that for simplicity, we assume that the header contains identifiers of three nodes, and in absence of failures, the two intermediate destination identifiers would be `null`. When messages are being re-routed to recover from a node failure, the intermediate destinations specify the escape edge of the child of the failed node that discovered the failure.

The protocol is formally defined below.

Protocol SNFR
// the node x has received the message $m(s|d_1|d_2|data)$.

`if` $m.d_1$ and $m.d_2$ are `null, do`
// Regular situation without any failures.
// Lookup the next hop to s in the primary lookup tables.
$y = nextHop(x, s)$
`if` y is `alive, do`
`forward` m `to` y
`else`
// Node y has failed.
// Route m via alternate path.
$e = \{p, q\}$ = my `escape edge` for s
$m.d_1 = p, m.d_2 = q$
$z = nextHop(x, p)$
`forward` m `to` z
`end-if`
`else`
// Intermediate destinations specified:
// Already in failure recovery mode
`if` $m.d_1 \neq$ `my ID, do`
// Need to forward the message to $m.d_1$
$w_1 = nextHop(x, m.d_1)$
`forward` m `to` w_1
`else`
// Local node is the first end point of the escape edge. Need to forward the message
// along the escape edge to $m.d_2$. Strip off the intermediate destination identifiers.
$w_2 = m.d_2$ // need to use the direct edge $\{m.d_1, m.d_2\}$
$m.d_1 =$`null`
$m.d_2 =$`null`
`forward` m `to` w_2
`end-if`
End Protocol SNFR

The recovery path for x_i computed by the SNFR algorithm may pass through the subtrees of multiple children of x before leaving the subtree of x. Once the path enters the subtree of a child $x_j \in \mathcal{C}_x$ at a node u, it becomes a *regular* message that the node u needs to forward towards s. The message is routed along the shortest path from u to s, as defined by $\mathcal{T}_s$. When the node x_j sees that the node x is not reachable, it re-routes the message along its alternate path (via its escape edge identified to escape from the failure of x). Interestingly, if x has recovered since the time x_i had noticed its failure, and initiated the re-routing, x_j simply forwards the message across to x, and the recovery path of x_i gets *short circuited*.

We also considered a few other versions of this basic protocol, but do not describe them in detail

here due to space limitations. In one version, we can reduce the extra space requirement in the message headers from two node identifiers to one at the cost of increased space requirements on the nodes themselves. A different version of the protocol wraps the original message into another message which is sent to the first end point of the escape edge, with information about the other end point in the message body. In this approach, one may need to split the original message into multiple such messages which will need to be assembled at the second end point of the escape edge back into the original message which can then be forwarded to the original destination.

4 Simulation Results and Comparisons

In this section, we present the simulation results for our algorithm, and compare the lengths of the recovery paths generated by our algorithm to the theoretically optimal paths. We also compare the lengths of our recovery paths, and the running time of our algorithm to those of [14]. In the implementation of our algorithm, we have used *standard* data structures (e.g. binary heaps instead of Fibonacci heap[4]: binary heaps suffer from a linear-time merge/meld operation as opposed to constant time for the latter). Consequently, our algorithms have the potential to produce much better running times than what we report.

4.1 Graph Generation

We ran our simulations on randomly generated graphs, with varying the following parameters: (a) Number of nodes, and (b) Average degree of a node. The edge weights are randomly generated numbers between 100 and 1000. In order to guarantee that the graph is *2-node-connected* (biconnected), we ensure that the generated graph contains a *Hamiltonian cycle*[5]. Finally, for each set of these parameters, we simulate our algorithm on multiple random graphs to compute the *average* value of the of a *metric* for the parameter set.

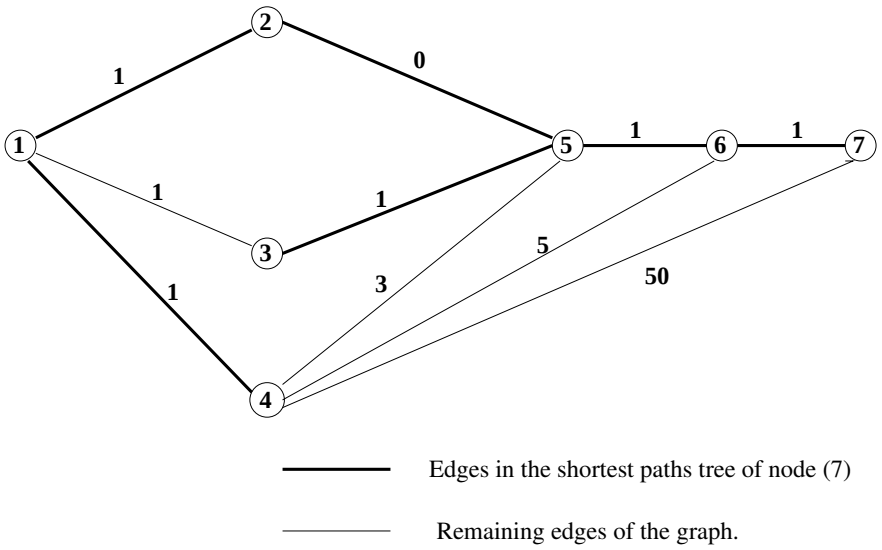
The algorithms have been implemented in the Java 5.0 [7] programming language (1.5.0.12 patch), and we ran them on an Intel machine with Pentium IV 3.06GHz processor and 2GB RAM running Red Hat Enterprise Linux 3.2.

4.2 Comparisons

We borrow the definition of *stretch factor* from [14]. Strech factor is defined as the ratio of the lengths of recovery paths generated by our algorithm to the lengths of the theoretically optimal paths. The optimal recovery path lengths when a node x fails are computed by recomputing the shortest paths tree of s in the graph $G(V \backslash x, E \backslash E_x)$.

Consider a network as in Figure 3, and we need to compute alternate paths for messages destined to node (7). The thicker edges form the original shortest paths tree of the node (7), and in absence of any failures, messages are routed along these links. The thinner edges represent the other links in the network. In their approach, the authors of [14] build a set of *key nodes* for each edge of the graph, and if a node receives a message for destination s via an edge e which is *unusual* for the destination (i.e. an edge via which it would never receive a message destined for s), the messages are routed along a path computed in the graph G after deleting *all* the key nodes for the edge e. A node p belongs to the key nodes set of edge e if the shortest path to s of the parent node of p (in

[5] we add edges to our graph if required in order to ensure that the graph contains a the Hamiltonian cycle



Edge {1,2} has key nodes:

5 – since path of node (2) uses edge (2,1) if (5) has failed.

6 – since path of node (5) uses edge (2,1) if (6) has failed.

Figure 3. Zhong et. al.'s algorithm[14] misses some good paths

$\mathcal{T}_s$) goes via the edge e when the node p fails. In our example, the key nodes for the edge $\{2, 5\}$ include nodes (5) and (6) since when the node (5) fails, the path from node (2) to node (7) uses the edge $\{2, 1\}$; and when the node (6) fails, the path from node (5) to node (7) uses the edge $\{2, 1\}$. Consequently, their algorithm results in longer paths when there are multiple key nodes for an edge. Our algorithm wins in such cases since we don't consider more nodes to have failed than reality. E.g. When the node (5) fails, the protocol of [14] resorts to routing the messages from (2) to (7) along the path $2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ which has a total weight of 52 units. In this case, our algorithm uses the optimal recovery path $2 \rightarrow 1 \rightarrow 4 \rightarrow 6 \rightarrow 7$, which has a total weight of 8 units. However, in some cases, there is only a single key node for an edge, which happens to be the actual node that fails. Our algorithm loses in such cases since the algorithm of [14] recomputes the shortest paths from scratch after deleting the failed node.

Computation of the key nodes is an expensive task which dominates the time complexity of the algorithm of [14]. Though the authors haven't presented a detailed analysis of their algorithm (probably because of space limitations), from our analysis, their algorithm needs at least $\Omega(mn \log n)$ time[6].

Another signification difference between the algorithm of [14] and our's is that their approach involves precomputing and storing an $O(n)$ size routing table per edge at the routers. The routing table is based on the key nodes for that edge, and thus differs across all edges of the router. Consequently, their *space* requirement is also higher than our's - we need to store additional information of only a single edge[7] (the escape edge for the node to escape the failure of it's parent) at each node

[6]For each node v in the graph, they need to compute the shortest path from the node $u = parent(v)$ to s in the graph $G(V \backslash v, E \backslash E_v)$. There are some more non-trivial computations besides this.

[7]Note the space requirements being discussed here for both algorithms is *per destination*.

(router).

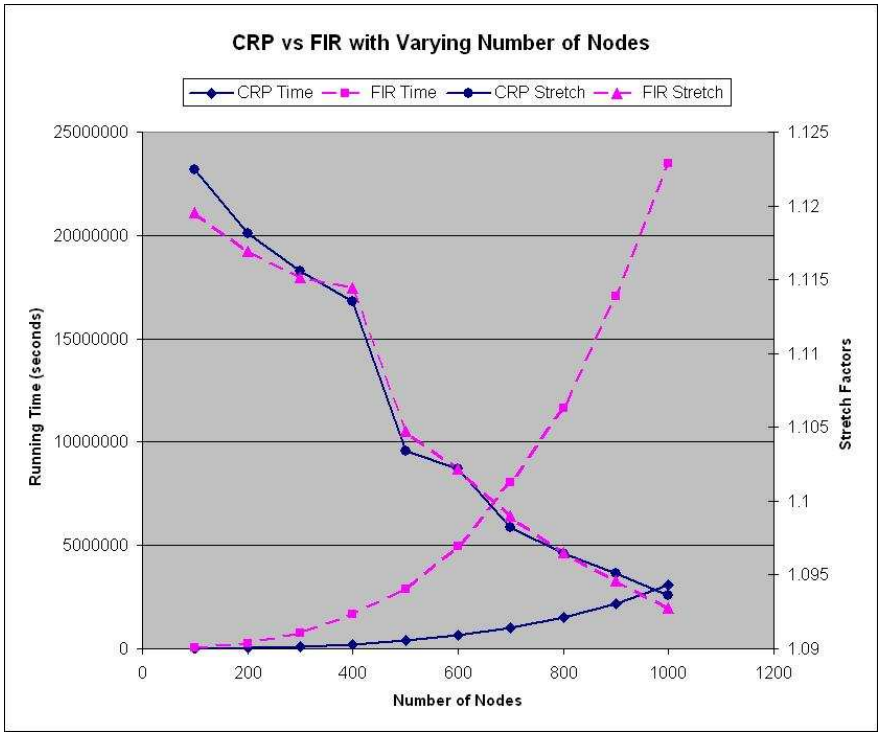


Figure 4. Performance with varying number of graph nodes

Figures [4,5] compare the performance of our algorithm (CRP) to that of [14] (FIR). The plots for the running times of our algorithm and that of [14] fall in line with the theoretical analysis that our algorithms are faster by an order of magnitude than those of [14]. Interestingly, the stretch factors of the two algorithms are very close for most of the cases. However, for reasons discussed above, there are cases where our algorithm's stretch factors are better than those of [14], as well as when they are worse. The metrics are plotted against the variation in (1) the number of nodes (Figure [4]), and (2) the average degree of the nodes (Figure [5]). The average degree of a node is fixed at 15 for the cases where we vary the number of nodes (Figure [4]), and the number of nodes is fixed at 200 for the cases where we plot the impact of varying average node degree (Figure [5]).

As expected, the stretch factors *improve* as the number of nodes increase. Our algorithm falls behind in finding the optimal paths in cases when the recovery path passes through the subtrees of multiple siblings. Instead of finding the best *exit* point out of the subtree, in order to keep the protocol simple and the paths well *structured*, our paths go to the root of the subtree and then follow its alternate path beyond that. These paths are formed using the blue edges. Paths discovered using a node's green edges are optimal such paths. In other words, if most of the edges of a node are green, our algorithm is more likely to find paths close to the optimal ones. Since the average degree of the nodes is kept fixed in these simulations, increasing the number of nodes increases the probability of the edges being green. A similar logic explains the plots in Figure [5]. When the number of nodes is fixed, increasing the average degree of a node results in an increase in the number of green edges for the nodes.

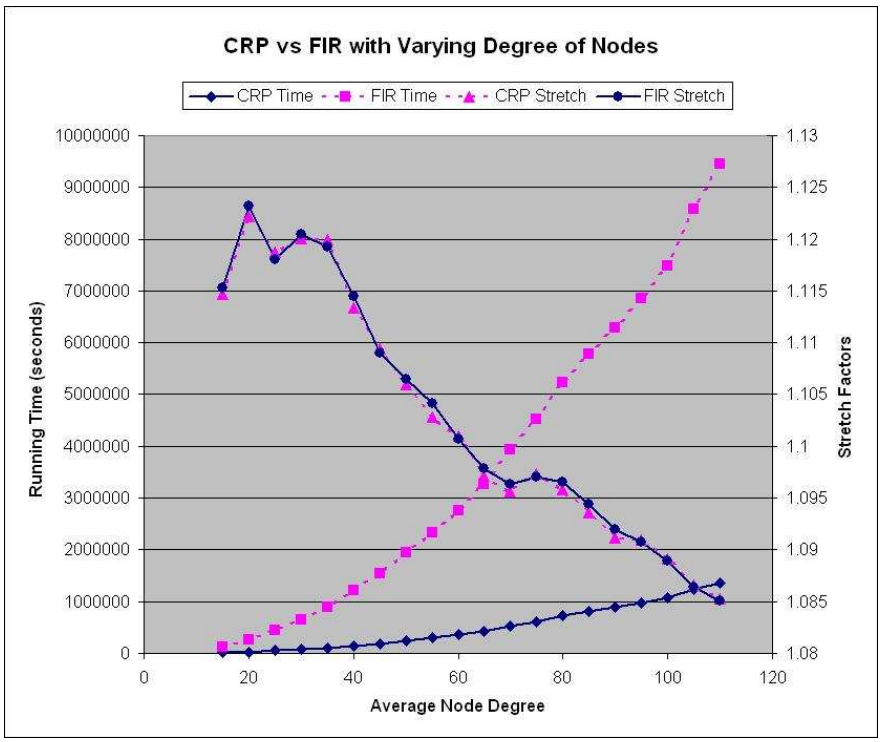


Figure 5. Performance with varying average degree of nodes

5 Concluding Remarks

In this paper we have presented an efficient algorithm for the SNFR problem, and developed protocols for dealing with transient single node failures in communication networks. Via simulation results, we show that our algorithms are much faster than those of [14], while the stretch factor of our paths are usually better or comparable.

The approach of [14] had previously been applied to the *link* version of the problem, and similar algorithms and protocols have been developed. See [9, 11] for further details on these. Interestingly the Single Link Failure Recovery (SLFR) algorithm presented by Bhosle and Gonzalez [1] computes *optimal* recovery paths, in *optimal* $O(m + n \log n)$ time. Protocols reported in this paper can be generalized to the link version of the problem based on the SLFR algorithms of [1]. These link failure recovery protocols would also work by passing the information about the escape edge in the message headers. However, it is important to note that unlike the recovery paths generated for the node version, the recovery paths used by these protocols would be *optimal*, since they use the exact paths generated by the optimal SLFR algorithm. Also, the SLFR algorithm is faster than those of [9, 11] by at least an order of magnitude.

The *single* node algorithms and protocols presented here can also be generalized to specific types of multiple node versions of the problems. One of these versions is the case where two nodes that are adjacent to each other in the shortest paths tree $\mathcal{T}_s$. Another version that can be handled is the failure of a node and all its neighbors in $\mathcal{T}_s$. These two types of multiple node failures are said to be *base* groups. A more general version of multiple node failures that can be handled is a group of nodes which can be iteratively built from a base group by adding a node that is adjacent to some

node in the base group. For instance, a node and all of its 2-hop neighbors forms such a group - we start with a single node, and grow the group by adding a node adjacent to a node in the group. Due to space limitations, we don't discuss these generalizations in more detail.

The *directed* version of the SNFR problem, where one needs to find the *optimal* (shortest) recovery paths can be shown to have a lower bound of $\Omega(min(m\sqrt{n}, n^2))$ using a construction similar to those used for proving the same lower bound on the directed version of SLFR[1] and replacement paths[6] problems. The bound holds under the *path comparison* model of [8] for shortest paths algorithms.

6 Acknowledgement

We thank Dr. Raju Rangaswami for reviewing an initial draft of this paper that greatly enhanced the readability of the same.

References

[1] A. M. Bhosle and T. F. Gonzalez. Algorithms for single link failure recovery and related problems. *J. of Graph Alg. and Appl.*, pages 8(3):275-294, 2004.

[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 *30^{th} ACM STOC*, pages 279-288. ACM Press, 1998.

[3] E. W. Dijkstra. A note on two problems in connection with graphs. In *Numerische Mathematik*, pages 1:269-271, 1959.

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

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

[6] J. Hershberger, S. Suri, and A. M. Bhosle. On the difficulty of some shortest path problems. *ACM Transactions on Algorithms*, 3(1), 2007.

[7] Sun Microsystems Inc. Java 5.0 programming language. *http://java.sun.com/j2se/1.5.0/*, 2007.

[8] 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.

[9] S. Lee, Y. Yu, S. Nelakuditi, Z.-L. Zhang, and C.-N. Chuah. Proactive vs reactive approaches to failure resilient routing. *In Proc. of IEEE INFOCOM*, 2004.

[10] A. Markopulu, G. Iannaccone, S. Bhattacharya, C. Chuah, and C. Diot. Characterization of failures in an ip backbone. *In Proc. of IEEE INFOCOM*, 2004.

[11] S. Nelakuditi, S. Lee, Y. Yu, Z.-L. Zhang, and C.-N. Chuah. Fast local rerouting for handling transient link failures. *IEEE/ACM Trans. on Networking*, pages 15(2):359s–372, 2007.

[12] 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.

[13] R. Rivest T. Cormen, C Leiserson and C. Stein. *Introduction to Algorithms*. McGraw Hill, 2001.

[14] Z. Zhong, S. Nelakuditi, Y. Yu, S. Lee, J. Wang, and C.-N. Chuah. Failure inferencing based fast rerouting for handling transient link and node failures. *In Proc. of IEEE INFOCOM*, pages 4: 2859-2863, 2005.