DISTRIBUTED ALGORITHMS FOR COMPUTING ALTERNATE PATHS AVOIDING FAILED NODES AND LINKS

Amit M. Bhosle*and Teofilo F. Gonzalez Department of Computer Science University of California, Santa Barbara, CA 93106 bhosle@alumni.cs.ucsb.edu, teo@cs.ucsb.edu

ABSTRACT

A recent study characterizing failures in computer networks shows that transient single element (node/link) failures are the dominant failures in large communication networks like the Internet. Thus, having the routing paths globally recomputed on a failure does not pay off since the failed element recovers fairly quickly, and the recomputed routing paths need to be discarded. In this paper, we present the first distributed algorithm that computes the alternate paths required by some *proactive recovery schemes* for handling transient failures. Our algorithm computes paths that avoid a failed node, and provides an alternate path to a particular destination from an upstream neighbor of the failed node. With minor modifications, we can have the algorithm compute alternate paths that avoid a failed link as well. To the best of our knowledge all previous algorithms proposed for computing alternate paths are centralized, and need complete information of the network graph as input to the algorithm.

KEY WORDS

Distributed Algorithms, Computer Network Management, Network Reliability, Routing Protocols.

1 Introduction

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, T_s , specifies the fastest way of 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 nodes¹ in the network may be susceptible to transient faults. These are sporadic faults of at most one node at a time that last for a relatively

short period of time. This type of situation has been studied in the past [1, 2, 7, 10, 11, 12] because it represents most of the node failures occurring in networks. Single node failures represent more than 85% of all node failures [8]. Also, these node failures are usually transient, with 46% lasting less than a minute, and 86% lasting less than 10 minutes [8]. 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 networks, and 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.

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 adjacent to a node that has 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. In this paper, we present an efficient distributed algorithm to compute such paths. Also, our algorithm can be generalized to solve some other problems related to finding alternate paths or edges.

A distributed algorithm for computing the alternate paths is particularly useful if the routing tables themselves are computed by a distributed algorithm since it takes away the need to have a centralized view of the entire network

^{*}Currently at Amazon.com, 605 5th Ave. S., Seattle, WA - 98144 ¹The nodes are *single*- or *multi*-processor computers

graph. Centralized algorithms inherently suffer from the overhead on the network administrator to put together (or source and verify) a consistent snapshot of the system, in order to feed it to the algorithm. This is followed by the need to deploy the output generated by the algorithm (e.g. alternate path routing tables) on the relevant computers (routers) in the system. Furthermore, centralized algorithms are typically resource intensive since a single computer needs to have enough memory and processing power to process a potentially huge network graph. Some other advantages of a distributed algorithm are reliability (no single points of failure), scalability and improved speed (computation time).

1.1 Related Work

A popular approach of tackling the issues related to transient failures of network elements is that of using *proactive recovery schemes*. These schemes typically work by precomputing alternate paths at the network setup time for the failure scenarios, and then using these alternate paths to re-route the traffic when the failure actually occurs. Also, the information of the failure is suppressed in the hope that the failure is transient and the failed element will recover shortly. The local rerouting based solutions proposed in [2, 7, 10, 11, 12] fall into this category.

Zhang, et. al. [12] present protocols based on local re-routing for dealing with transient single node failures. They demonstrate via simulations that the recovery paths computed by their algorithm are usually within 15% of the theoretically optimal alternate paths.

Wang and Gao's Backup Route Aware Protocol (BRAP) [11] also uses some precomputed backup routes in order to handle transient single *link* failures. One problem central to their solution asks for the availability of *reverse paths* at each node. However, they do not discuss the computation of these reverse paths. As we discuss later, the alternate paths that our algorithm computes qualify as the reverse paths required by the BRAP protocol of [11].

Slosiar and Latin [10] studied the single *link* failure recovery problem and presented an $O(n^3)$ time for computing the link-avoiding alternate paths. A faster algorithm, with a running time of $O(m+n\log n)$ for this problem was presented in [1].

1.2 Preliminaries

Our communication network is modeled by an edgeweighted 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.

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 a 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 $\mathcal{C}_x = \{x_1, x_2, \dots, x_{k_x}\}$ 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 . The parent node, p, of a node c is sometimes referred to as a *primary neighbor* or *primary router* of c, while c is referred to as an *upstream neighbor* or *upstream router* of p. The children of a particular node are said to be *siblings* of each other.

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

1.3 Problem Definition

The Single Node Failure Recovery problem is formally defined in [2] 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 $\mathcal{C}_x = \{x_1, x_2, \dots, x_{k_x}\}$ denotes the set of *children* of 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 $G = (V \setminus \{x\}, E \setminus E_x)$, where E_x is the set of edges adjacent to x.

In other words, for each node x in the graph, we are interested in finding alternate paths from each of its children in \mathcal{T}_s to the node s when the node x fails. Note that the problem is not well defined when node s fails.

The above definition of alternate paths matches that in [11] for *reverse paths*: for each node $x \in G(V)$, find a path from x to the node s that does not use the primary neighbor (parent node) y of x in \mathcal{T}_s .

1.4 Main Results

Our main result is an efficient distributed algorithm for the SNFR problem. Our algorithm requires O(m + n) messages to be transmitted among the nodes (routers), and has a space complexity of O(m+n) across all nodes in the network (this, being asymptotically equal to the size of the entire network graph, is asymptotically optimal). The space requirement at any single node is linearly proportional to the number of children (the node's degree) and the number of siblings that the node has in the shortest paths tree of the destination s. When used for multiple sink nodes in the network, the space complexity at each node is bounded by its total number of children and siblings across the shortest paths trees of all the sink nodes. Note that even though this is only bounded by $O(n^2)$ in theory (since each node in the network can be a sink, and a node can theoretically have O(n) children), it is much smaller in practice (O(n): for n sink nodes, as average node degree in shortest paths trees is usually within 20-40 even for n as high as a few 1000s).

Our algorithm is based on a request-response model, and does not require any *global coordination* among the nodes.

To the best of our knowledge, this is the first completely decentralized and distributed algorithm for computing alternate paths. All previous algorithms, including those presented in [1, 2, 7, 10, 11, 12] are centralized algorithms that work using the information of the entire network graph as input to the algorithms.

Furthermore, our algorithm can be generalized to solve other similar problems. In particular, we can derive distributed algorithms for: the single link failure recovery problem studied in [1, 10], minimum spanning trees sensitivity problem [4] and the detour-critical edge problem [9]. The cited papers present centralized algorithms for the respective problems.

2 Key Properties of the Alternate Paths

We now describe the key properties of the alternate paths to a particular destination that can be used by a node in the event of its parent node's failure. These same principles have been used in the design of the centralized algorithm in [2]. However, for completeness, we discuss them briefly here.



Figure 1. Recovering from the failure of x: Constructing the recovery graph \mathcal{R}_x

Figure 1(a) illustrates a scenario of a single node failure. In this case, the node x has failed, and we need to find alternate paths to s from each $x_i \in C_x$. When a node fails, the shortest paths tree of s, T_s , gets split into $k_x + 1$ components - one containing the source node s and each of the remaining ones containing the subtree of a child $x_i \in C_x$.

Notice that the edge $\{g_p, g_q\}$, 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 $p_G(x_j, g_p) \rightarrow \{g_p, g_q\} \rightarrow p_G(g_q, s)$. 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 $x_j \in C_x$ to a node outside the subtree of x) can be used by $x_j \in C_x$ to *escape* the failure of node x. Such edges are called *green* edges. For example, the edge $\{g_p, g_q\}$ is a green edge.

Next, consider the edge $\{b_u, b_v\}$ 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 $\{b_u, b_v\}$ and $\{g_p, g_q\}$ 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 $(p_G(x_i, b_u) \rightarrow \{b_u, b_v\} \rightarrow p_G(b_v, x_j))$, 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 a sibling x_j for some $i \neq j$) are called *blue* edges. $\{b_p, b_q\}$ is another blue edge and can be used by the node x_1 to recover from the failure of x.

Note that edges like $\{r_a, r_b\}$ and $\{b_v, g_p\}$ with both end points within the subtree of the same child of x do not help any of the nodes in C_x to find a recovery path from the failure of node x. We do not consider such *red* 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 quite 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 these edges and eliminate useless (red) ones efficiently.

We now describe the construction of a new graph \mathcal{R}_x , called the *recovery graph* of x, which will be used to compute recovery paths for the elements of \mathcal{C}_x when the node x fails. 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 \mathcal{R}_x has $k_x + 1$ nodes, where $k_x = |\mathcal{C}_x|$. A special node, s_x , represents in \mathcal{R}_x , the 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 \mathcal{R}_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 y_i and s_x . 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 y_i and y_i .

Note that the weight of the edges added to \mathcal{R}_x need not be the same as the weight of the corresponding green or blue edges in G = (V, E). The weights assigned to the edges in \mathcal{R}_x should take into account the weight of the actual subpath in G corresponding to the edge in \mathcal{R}_x . As long as the weights of edges in \mathcal{R}_x don't change with x, or can be determined locally by the node, they can be directly used in our algorithm. The candidate recovery path of x_j that uses the green edge $e = \{u, v\}$ has total cost given by:

$$greenWeight(e) = d_G(x_j, u) + cost(u, v) + d_G(v, s)$$
(1)

This weight captures the weight of the actual subpath in G corresponding to the edge added to \mathcal{R}_x . However, since the weight given by equation (1) for an edge depends on the node x_j whose recovery path is being computed, it will typically be different in each \mathcal{R}_x in which e appears as a green edge. The following weight function is more efficient since it remains constant across all \mathcal{R}_x graphs that e is part of.

$$greenWeight(e)$$

$$= d_G(s, x_j) + d_G(x_j, u) + cost(u, v) + d_G(v, s)$$

$$= d_G(s, u) + cost(u, v) + d_G(v, s)$$
(2)

Note that the correct weight (as defined by equation (1)) to be used for an \mathcal{R}_x can be derived by the node x from the weight function defined above by subtracting $d_G(s, x_j) = d_G(s, x) + cost(x, x_j)$. Also, the green edge with an end point in the subtree of x_j with the minimum greenWeight remains the same, immaterial of the green-Weight function (equations (1) or (2)) used since equation (2) basically adds the value $d_G(s, x_j)$ to all such edges.

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 $b = \{p, q\}$ is the blue edge connecting the subtrees of x_i and x_j the length of the subpath from x_i to x_j is:

$$blueWeight(b) = d_G(x_i, p) + cost(p, q) + d_G(q, x_i)$$
(3)

We assign this weight to the edge corresponding to the blue edge $\{p, q\}$ that is added in \mathcal{R}_x between y_i and y_j .

Note that if w is the nearest common ancestor of the two end points u and v of and edge e = (u, v), e is a green edge in the \mathcal{R} graphs for all nodes on path between w and u, and w and v (excluding u, v and w: it is a blue edge in \mathcal{R}_w , and is unusable in \mathcal{R}_u and \mathcal{R}_v since a node z is deemed to have failed while constructing \mathcal{R}_z). Assuming that a node can determine whether an edge is blue or green in its recovery graph (we discuss this in detail in the next section), it is easy to see that it can derive the edge's blue weight from its green weight:

$$blueWeight(e) = greenWeight(e) - (2 \cdot d_G(s, w) + cost(w, w_u) + cost(w, w_v))$$
(4)

where w_u and w_v are respectively the child nodes of wwhose subtrees contain the nodes u and v. Information about all terms being subtracted is available locally at w, and consequently, the greenWeight and blueWeight values for an edge can be computed/derived using information local to the node w. 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 (with ties being broken arbitrarily) and ignore the rest. Similarly, if there are multiple edges between the subtrees of two siblings x_i and x_j , we retain the one which offers the cheapest alternate path.

The construction of our graph \mathcal{R}_x is now complete. Computing the shortest paths tree of s_x in \mathcal{R}_x provides enough information to compute the recovery paths for all nodes $x_i \in \mathcal{C}_x$ when x fails.

Note that any edge e = (u, v) acts as a blue edge in at most one \mathcal{R}_x : that of the nearest-common-ancestor of u and v. Also, any node $c \in G(V)$ belongs to exactly one \mathcal{R}_x : that of its parent in \mathcal{T}_s . As we discuss later, the space requirement at any node is linearly proportional to the number of children and the number of siblings that it has.

Figure 1 illustrates the construction of \mathcal{R}_x used to compute the recovery paths from the node $x_i \in \mathcal{C}_x$ to the node s when the node x has failed. In this simple example, the path from y_i to s_x is $y_i \rightsquigarrow y_j \rightsquigarrow s_x$. The corresponding recovery path for x_i is $p_G(x_i, b_u) \rightsquigarrow$ $\{b_u, b_v\} \rightsquigarrow p_G(b_v, x_j)$, followed by the recovery path of $x_j: p_G(x_j, g_p) \rightsquigarrow \{g_p, g_q\} \rightsquigarrow p_G(g_q, s)$.

3 A Distributed Algorithm for Computing the Alternate Paths

In this section, we use the basic principles of the alternate paths described earlier to design an efficient distributed algorithm for computing the alternate paths.

3.1 Computing the DFS Labels

Our distributed algorithm requires that each node in the shortest paths tree \mathcal{T}_s maintain its $dfsStart(\cdot)$ and $dfsEnd(\cdot)$ labels in accordance with how a depth-firstsearch (dfs) traversal of \mathcal{T}_s starts or ends at the node. Ref. [5] reports efficient distributed algorithms for this particular problem (of assigning labels to the nodes in a tree as dictated by a dfs traversal of the tree). The basic algorithm reported in Ref. [5], named Wake & Label_A, assigns dfs labels to the nodes in the range [1, n] in asymptotically optimal time and requires 3n messages to be exchanged between the nodes. We sketch the basic algorithm below.

The Wake & Label_A algorithm runs in three phases: wakeup, count, and allocation. In the first (wakeup) phase, which is a top-down phase, the root node sends a message to all of its child nodes asking them to report the number of nodes in their subtree (including themselves). The child nodes recursively pass on the message to their children. In the second (count) phase, which is a bottom-up phase, each node reports the size of its subtree to its parent node. The variants of the Wake & Label algorithms differ in the last phase (allocation) which deals with assigning the labels to the nodes of the tree. In the sim-

plest version, once the root node knows the value of n (the total number of nodes in the tree), knowing the size of the subtrees of each child node, it can split the range [1, n] disjointly among its children, and each child node recursively assigns a sub-range to its children (a child with c nodes in its subtree is assigned a range containing c values).

The reader is referred to Ref. [5] for the detailed description and analysis of the Wake & Label_A algorithm and its variants. For computing the $dfsStart(\cdot)$ and $dfsEnd(\cdot)$ labels required by our algorithm, the total range of these labels across all the nodes in T_s is [1, 2n], and a child with c children is assigned a range of 2c values. All other aspects of any of the dfs label assignment algorithms reported in Ref. [5] can be used as appropriate. Note that even though it is not explicitly mentioned in Ref. [5], the Wake & Label_A algorithm (including our modifications) can be implemented on a request-response model, without the need of any global clock for coordination across the nodes.

3.2 Collecting the Green and Blue Edges

Our algorithm requires that each node in the network maintains the following data-structures:

1. ParentBlueEdges List: The list of edges in the network graph which have one end point within the subtree of the node, and the other end point in the subtree of a sibling node. I.e. all edges from the node's subtree that are *blue* in the recovery graph \mathcal{R} of the node's parent.

2. ChildrenGreenEdges Map: A map that stores for each child node, the cheapest green edge with an end point in the child node's subtree. Recollect that a green edge of a node has the other end point outside the subtree of the node's parent.

We now discuss the details of this part of the algorithm for building the ParentBlueEdges and ChildrenGreenEdges data-structures. A procedure, CollectNonTreeEdges, triggers a protocol where each node recursively asks each of its children to forward it the non-tree edges that have an end point in the child's subtree. Each node processes all its own non-tree edges, and those forwarded by a child node. For processing a non-tree edge, a node uses the $dfsStart(\cdot)$ and $dfsEnd(\cdot)$ labels of the edge's two end points to decide whether the edge should be added to its ParentBlueEdges list or the ChildrenGreenEdges map. For an edge to be added to the ParentBlueEdges list, the edge should have exactly one end point in the node's subtree, while the other end point still be within the parent's subtree (but outside this node's subtree). For each edge that is forwarded by a child, the node updates the corresponding entry for the child in the ChildrenGreenEdges map if the newly forwarded edge is cheaper than the edge currently stored for the child. Finally, if at least one of the two end points of the edge lies outside this node's subtree, it forwards the information of the edge to the parent after updating its local data-structures. Otherwise, it simply discards the edge and does not forward it to its parent. The reason for this is that edges whose both end points belong to a node's subtree cannot serve as a blue or green edge in the recovery graph of the node's parent, and informing the parent about such an edge does not serve any purpose (if this node is the nearest-common-ancestor of the edge's two end points, the edge would be stored in the ParentBlueEdges lists at the two child nodes whose subtrees contain the edge's end points).

A child node invokes the procedure RecordNonTreeEdge defined below on its parent, with a message \mathcal{M} containing the following information associated with a non-tree edge e:

- $e = (p_1, p_2)$: The non-tree edge, with p_1 and p_2 as the end points.
- weight(e): Weight of the edge e.
- *senderId*: Id of this child node sending the message to the parent node.

Procedure RecordNonTreeEdge(\mathcal{M})

```
if (isMyDescendant(\mathcal{M}.p_1) AND
  isMyDescendant(\mathcal{M}, p_2)) do:
  // both end points in my
  // subtree: ignore
  return;
 fi
 // retrieve the current green
 // edge for this sender from
 // the ChildrenGreenEdges map
 Edge existing =
  CGE.get(\mathcal{M}.senderId);
 Edge edge = \mathcal{M}.edge;
 if (existing == null OR
  edge.weight < existing.weight), do:</pre>
  // if new or cheaper edge,
  // update our data-structure
  CGE.put(\mathcal{M}.senderId, edge);
 fi
 if (edgeIsBlueForParent(edge)), do:
  ParentBlueEdges.add(edge);
 fi
 // Reset the senderId,
 // and forward edge to parent
 \mathcal{M}.senderId = self.id;
 parent.RecordNonTreeEdge(\mathcal{M});
End RecordNonTreeEdge
```

The edgeIsBlueForParent method used above determines whether or not an edge is blue for this node's parent. This can be determined easily if the node knows its parent's $dfsStart(\cdot)$ and $dfsEnd(\cdot)$ labels. For efficiency, after the dfs label computation phase (AssignDfsLabels) is over, each node can query its parent for its labels, and store these locally. In some cases, these values can just be queried from the parent node as and when needed.

3.3 Computing the Alternate Paths to Recover from a Node's Failure

Once the edge propagation phase is over, part of the information required to construct \mathcal{R}_x , the recovery graph of x, is available at the node x, and the remaining is available at the children of x. In particular, x has the information about the nodes of \mathcal{R}_x and the green edges of \mathcal{R}_x , while the children of x have the information of the blue edges of \mathcal{R}_x .

Conceptually, x can construct the entire graph \mathcal{R}_x locally, and compute the shortest paths tree of s_x . This process would result in a space complexity of $O(m_x + n_x)$ at node x, where m_x and n_x denote the number of edges and nodes in \mathcal{R}_x respectively. Note that m_x can be as large as $O(n_x^2) = O(|\mathcal{C}_x|^2)$. In order to keep the space requirement low, the shortest paths tree, T_{s_x} , of s_x is built incrementally, by looking at the edges of \mathcal{R}_x only when they are needed. Essentially, we use the edges exactly in the order dictated by the Dijkstra's shortest paths algorithm[3]. x initially builds \mathcal{R}_x using the information it locally has: the $k_x + 1$ nodes, and the green edge from y_i to s_x for $1 \leq i \leq k_x$ (if the ChildrenGreenEdges map has an entry for x_i). x maintains a priority queue data structure, candidates, which initially has an entry for each y_i , with a priority² equal to the weight of the edge between s_x and y_i^3 . The remaining steps of the algorithm are as follows.

- 1. While there are more entries in candidates, execute steps 2 - 4.
- 2. Delete entry from candidates with highest priority.
- 3. Assign the priority value as the final distance (from s_x) for the node y_p associated with the queue entry.
- 4. Fetch the blue edges from child node x_p . For each blue edge thus retrieved, if it provides a shorter path to its other end point, say x_q , update the priority of the queue entry corresponding to y_q with this value.

Note that the blue edges stored at a child node x_p are retrieved only when they are needed by the algorithm, and that each node x needs space linearly proportional to its number of children, and the number of its siblings. For each sibling, a node needs to store at most one edge (which has the smallest blue weight) with an end point in its own subtree, and the other in the sibling's subtree. These edges are the blue edges that are added to the parent node's recovery graph. Using Fibonacci heaps[6] for the priority queue, T_{sx} can be computed in $O(m_x + n_x \log n_x)$ time.

4 Concluding Remarks

In this paper we have presented an efficient distributed algorithm for the computing alternate paths that avoid a failed node. To the best of our knowledge, this is the first completely decentralized algorithm that computes such alternate paths. Our algorithm can be generalized to solve other similar problems.

The paths computed by our algorithm are required by the single node failure recovery protocol of [2]. They also qualify as the *reverse paths* required by the BRAP protocol of [11]. Our distributed algorithm computes the exact same paths as those generated by the centralized algorithm of [2], and even though not optimal alternate paths, they are usually good - within 15% of the optimal for randomly generated graphs with 100 to 1000 nodes, and with an average node degree of up to 35. The reader is referred to [2] for further details about the simulations.

References

- A. M. Bhosle and T. F. Gonzalez. Algorithms for single link failure recovery and related problems. *J. Graph Algorithms Appl.*, 8(2):275–294, 2004.
- [2] A. M. Bhosle and T. F. Gonzalez. Efficient algorithms and routing protocols for handling transient single node failures. In 20th IASTED PDCS, 2008.
- [3] E. W. Dijkstra. A note on two problems in connection with graphs. In *Numerische Mathematik*, pages 1:269-271, 1959.
- [4] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Jr. C.*, 21(6):1184–1192, 1992.
- [5] P. Fraigniaud, A. Pelc, D. Peleg, and S. Perennes. Assigning labels in unknown anonymous networks (extended abstract). In *PODC*, 101–111, 2000.
- [6] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34:596-615, 1987.
- [7] S. Lee, et. al. Proactive vs reactive approaches to failure resilient routing. IEEE INFOCOM, 2004.
- [8] A. Markopulu, *et. al.* Characterization of failures in an ip backbone. *In Proc. of IEEE INFOCOM*, 2004.
- [9] E. Nardelli, G. Proietti, and P. Widmayer. Finding the detour-critical edge of a shortest path between two nodes. *Inf. Process. Lett.*, 67(1):51–54, 1998.
- [10] R. Slosiar and D. Latin. A polynomial-time algorithm for the establishment of primary and alternate paths in atm networks. In *IEEE INFOCOM*, 509–518, 2000.
- [11] F. Wang and L. Gao. A backup route aware routing protocol - fast recovery from transient routing failures. In *INFOCOM*, 2008.
- [12] Z. Zhong, et. al. Failure inferencing based fast rerouting for handling transient link and node failures. IEEE INFOCOM, 2859–2863, 2005.

²lower value implies higher priority

³ if no edge is present, a priority of ∞ is assigned