PAPER

Point-of-Failure Swap Rerouting: Computing The Optimal Swaps Distributively

Paola FLOCCHINI[†], **Antonio MESA ENRIQUES**[††], **Linda PAGLI**[†††], **Giuseppe PRENCIPE**[†††], ***and*** **Nicola SANTORO**[††], ***Nonmembers***

SUMMARY We consider the problem of computing the *optimal swap edges* of a shortest-path tree. This problem arises in designing systems that offer *point-of-failure shortest-path rerouting* service in presence of a single link failure: if the shortest path is not affected by the failed link, then the message will be delivered through that path; otherwise, the system will guarantee that, when the message reaches the node where the failure has occurred, the message will then be re-routed through the shortest-path to its destination. There exist highly efficient serial solutions for the problem, but unfortunately because of the structures they use, there is no known (nor foreseeable) efficient distributed implementation for them. A distributed protocol exists only for finding swap edges, not necessarily optimal ones.

We present two simple and efficient distributed algorithms for computing the optimal swap edges of a shortest-path tree. One algorithm uses messages containing a constant amount of information, while the other is tailored for systems that allow long messages. The amount of data transferred by the protocols is the same and depends on the structure of the shortest-path spanning-tree; it is no more, and sometimes significantly less, than the cost of constructing the shortest-path tree.

(1) Keywords.

Fault-Tolerant Routing, Point of Failure Rerouting, Shortest Path Spanning Tree, Weighted Graphs, Distributed Algorithms.

1. Introduction

1.1 The Framework and Previous Work

In systems using shortest-path routing tables, a single link failure is enough to interrupt the message transmission by disconnecting one or more shortest-path spanning trees. The on-line recomputation of an alternative path or of the entire new shortest path trees, rebuilding the routing tables accordingly, is rather expensive and causes long delays in the message's transmission [10], [15]. Some of these costs and delays could be reduced if the serial algorithms for dynamic graphs (e.g., those of [5]) could somehow be exploited. These serial algorithms pre-compute some useful information and store it in ad-hoc data structures. The difficulty is clearly in distributing the computation and the data structures, and more importantly in the distributed maintenance of this dynamic structure; these difficulties have not yet been successfully overcome (e.g., see [12]).

An alternative approach is to pre-compute additional information and use it to augment the shortest-path routing tables so to make them operate when a failure occurs. Examples of this approach are techniques (e.g., see [9]) of pre-computing and storing, in addition to the shortest-path routing tables, k other spanning trees for each destination [9]. A single additional tree would not be sufficient because it would need to be edge-disjoint with all the original shortest-path spanning-trees defining the routing tables, and such a tree might not exist; hence $k > 1$. The storage requirements are reasonable: for each destination, a node stores k links (one for each additional tree) in addition to the one in the fault-free shortest-path. The pre-computing can be done using any the existing distributed algorithms distributed algorithms for constructing possibly edge-disjoint spanning-trees or collection of disjoint paths (e.g., [1], [11]). However, the alternative routes do not satisfy any optimization criterion (such as shortest path) even in the case when only one link might be down.

To reduce the amount of communication and of storage, a new strategy has been recently proposed [4], [10], [13], [14], [16]. It starts from the idea of pre-computing, for each link in the tree, a single non-tree link (the *swap* edge) able to reconnect the network should the first fail. The strategy, called *point-of-failure swap rerouting* is simple: normal routing information will be used to route a message to its destination. If, however, the next hop is down, the message is first rerouted towards the swap edge; once this is crossed, normal routing will resume. This approach has several advantages. In particular, there is no need to broadcast a link failure and its subsequent reactivation (if any). Furthermore, it requires only a single item of additional information (the swap) to be added to each item of the shortest-path routing tables. This fact makes the approach particularly suited for systems which employ static routing table, or where memory requirements are stringent.

The problem of determining a swap edge for each link in the shortest path trees has been investigated by Ito *et al* [10], who presented an efficient sequential algorithm that can, possibly, be efficiently implemented in

[†]The author is with the University of Ottawa, Canada.
[††]The author is with Carleton University, Canada.
[†††]The author is with University of Pisa, Italy. **Contact author:** prencipe@di.unipi.it

a distributed setting. A basic drawback of their solution is that swap edges they determine are arbitrary, i.e. they do not have any particular property. This means that, if the failure occurs, the system does not make any guarantee other than message delivery. Although acceptable in some contexts, this level of service might not be tolerable in general.

Clearly, some swap edges are preferable to others and, choosing appropriately, the routing system can be made to offer a higher level of service. Consider for example the choice of swap edges such that if the shortest path is not affected by the failed link, then the message will be delivered through that path; otherwise, the system will guarantee that, when the message reaches the node where the failure has occurred, the message will then be re-routed to its destination through the *shortest-path*. Such a service is called *point-of-failure shortest-path rerouting* and the set of swap edges enabling it are called *optimal swap* edges.

The advantage of having optimal swap edges can be significant. In fact, experimental results [16] show that the tree obtained from using an optimal swap edge is very close to the new shortest-path spanning tree computed from scratch.

The problem of computing all the optimal swap edges for a shortest-path tree has been attacked by Nardelli, Proietti, and Widmayer [14]. They showed that the problem can be solved sequentially in $O(m \cdot \alpha(m,n))$ time, where $\alpha(m,n)$ is the functional inverse of Ackermann's function. This bound is achieved using Tarjan's sophisticated technique for union-find, which requires the construction of *transmuters* [17]. Unfortunately, there is currently no efficient distributed implementation of this sequential technique; since in a distributed network setting the construction of transmuters requires complete global network information at some node, it is doubtful whether this approach can be efficiently implemented in a distributed setting.

Summarizing, the problem of computing all the optimal swap edges of a shortest-path tree is an interesting graph-theoretic problem and a crucial component to implement a point-of-failure shortest-path rerouting strategy. Currently, there is no distributed solution (in [10], the problem is posed but no solution given). Clearly any such a solution should not add significantly to the overall cost of constructing the final routing tables. In particular, the computation should not require more messages (at least in order of magnitude) than those used to construct the shortest-path tree.

1.2 Our Contribution

In this paper we present an efficient distributed solution to the optimal swap edges problem.

Given a shortest-path spanning-tree T_r, the proposed protocol determines at each node x the optimal swap edge for $e_x = (x, p(x))$, with $p(x)$ the parent of x. The algorithm uses $O(n_r^*)$ messages of *constant* size, where n_r^* is the size of the transitive closure of $T_r \setminus \{r\}$; observe that $0 \leq n_r^* \leq (n-1)(n-2)/2$.

If longer messages are allowed, the same strategy can be modified to construct a different algorithm that uses only $O(n)$ such messages.

Providing a uniform comparison between protocols using different sized messages, the *data complexity* of a protocol measures the total amount of data exchanged during the execution; in our context, a node, an edge, a label, a weight, and a distance are each a unit of data. Both algorithms have an overall data complexity of $O(n_r^*)$.

Notice that this cost is always less, and oftentimes substantially so, than the cost of constructing a shortest-path spanning-tree. We actually conjecture that such a cost is *optimal*.

Further notice that the information assumed available by our algorithms can be acquired during the shortest-path spanning-tree construction, without increasing the order of magnitude of the message and information complexity of that process. Should this information not be provided, it can be easily acquired with an $O(m)$ data complexity.

The paper is organized as follows. In the next section we introduce some definitions and terminology. The new distributed algorithm for constructing all the optimal swap edges for a given shortest-path spanning-tree using constant-size messages is described and analyzed in Section 3. In Section 4, we present a more efficient algorithm for systems allowing long messages. The concluding remarks and open problems are in Section 6.

2. Definitions and Terminology

Let $G = (V, E)$ be a simple undirected graph, with $n = |V|$ vertices and $m = |E|$ edges. A *subgraph* $G' = (V', E')$ of G is any graph where $V' \subseteq V$ and $E' \subseteq E$. If $V' \equiv V$, G' is a *spanning* subgraph. A *path* $P = (V_p, E_p)$ is a subgraph of G, such that $V_p = \{v_1, \ldots, v_s\} | v_i \neq v_j$, for $i \neq j$, and $(v_i, v_{i+1}) \in E_p$, for $1 \leq i \leq s-1$. If $v_1 = v_s$ then P is a *cycle*. A graph G is *connected* if, for each pair $\{v_i, v_j\}$ of its vertices, there exists a path connecting them. A graph G is *biconnected* if, after the removal of anyone of its edges it remains connected. A *tree* is a connected graph with no cycles.

A non negative real value called *weight* (or *length*) and denoted by $|e|$ is associated to each edge e in G. Given a path P, the length of the path is the sum of the lengths of its edges. The *distance* $d_{G'}(x, y)$ between two vertices x and y in a connected subgraph G' of G, is the length of the shortest path from x to y in G'. For simplicity, in the following we will denote $d_G(x, y)$ simply by $d(x, y)$.

For a given vertex r, called *source*, the *shortest*

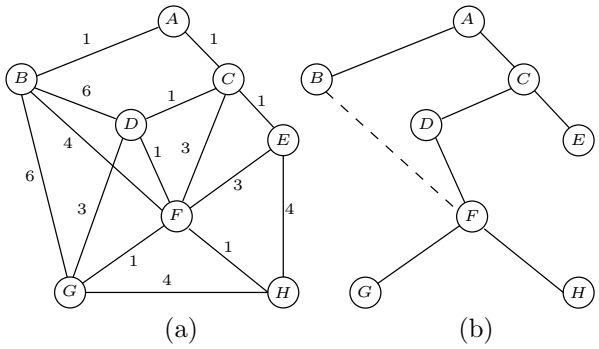

Fig. 1 (a) A biconnected weighted graph G. (b) The shortest-path spanning tree T_A rooted in A; the dotted edge (F, B) is the optimal swap edge for (C, A).

path tree (SPT) of r is the spanning tree T_r rooted at r such that the path in T_r from r to any node v is the shortest possible one; i.e., $\forall x \in V\ d_{T_r}(x, r) = d(x, r)$.

The removal of any edge e of T_r will disconnect T_r into two subtrees. If G is biconnected, there will always be at least an edge $e' \in E(G) \setminus E(T_r)$ that will join the two disconnected subtrees forming a new spanning tree T' of G. Any such an edge is called a *swap* edge for e; let $S_r(e)$ denote the set of swap edges for e.

An *optimal swap edge* (or *bridge*) for $e = (u, v)$ is any swap edge for e such that the distance from u to the source r in the new tree T' is minimized; more precisely, an optimal swap edge for $e = (u, v)$ is a swap edge $e' = (u', v') \in S_r(e)$ such that $d_{T'}(u, r) = d_{T_r}(u, u') + |(u', v')| + d_{T_r}(v', r)$ is minimum. As an example, consider the biconnected weighted graph G shown in Figure (1.a), and the shortest-path spanning tree T_A rooted in A shown in Figure in (1.b). It is easy to verify that the optimal swap edge for (C, A) is (F, B).

The *optimal swap edges problem* for $T(r)$ is the problem of determining an optimal swap edge for each edge in T_r.

As already mentioned, a sequential algorithm solving this problem was given in [14]. We are interested in the distributed solution of the optimal swap edge problem. We consider a *distributed computing system* with communication topology G. Each computational entity x is located at a node of G, has local processing and storage capabilities, has a distinct label $\lambda_x(e)$ from a totally ordered set associated to each of its incident edges e, knows the weight of its incident edges, and can communicate with its neighboring entities by transmission of bounded sequence of bits called messages. The communication time includes processing, queueing, and transmission delays, and it is finite but otherwise unpredictable. In other words, the system is *asynchronous*. All the entities execute the same set of rules, called distributed algorithm.

In the following, when no ambiguity arises, we will use the terms entity, node and vertex as equivalent; analogously, we will use the terms link, arc and edge interchangeably.

3. Computing All Optimal Swap Edges

We now present a solution to the problem of distributively computing all optimal swap edges for a given shortest-path spanning tree T_r.

3.1 Basic Properties and Tools

In our algorithm we make use of some known properties of rooted trees. In T_r each node except the root r has a unique *parent*, and each edge connects a node to its parent.

Property 1: The partial order induced by the relation *parent* has dimension at most 2.

Consider in fact the labelling $\alpha : V \to \{1, \ldots, n\}^2$ defined as follows. Given T_r, for $x \in V$ let $\alpha(x) = (a, b)$, where a is the numbering of x in the *preorder* traversal[†] of T_r; and b is the numbering of x in the *inverted preorder* traversal of T_r, i.e., when the order of the visit of the children is inverted. The labels associated to the nodes in the tree of Figure 1.b are shown in Figure 2.a.

Let $T_r[x]$ denote the subtree of T_r rooted in x. Any node y in the subtree $T_r[x]$ is said to be a *descendant* of x. Let $Desc_r(x)$ be the set of the descendants of x in T_r; note that, by definition, $x \in Desc_r(x)$. Interestingly, the *lexicographic order* $\succ$ between the labels assigned by α completely characterizes the *descendant* relationship in a rooted tree:

Property 2: A node y is descendant of a node x in T_r *if and only if* $\alpha(y) \succeq \alpha(x)$.

Property 2 can be easily verified and it is known as a folklore method to check relationships among nodes in trees.

Furthermore, there exists a simple relationship between swap edges and the descendants.

Property 3: An edge $(u, v) \in E \setminus E(T_r)$ is a swap for $e_x \in E(T_r)$ *if and only if* only one of u and v (but not both) is in $Desc_r(x)$.

For brevity, we will denote the set $S_r(e_x)$ of all swap edges for e_x simply by $S(x)$, and by $InS(x) \subseteq S(x)$ the set of those that are incident on x. The last useful property states that the swap edges for e_x consists only of all the swap edges incident to x and to its descendants.

Property 4: For all $x \in V$, $S(x) = \bigcup_{y \in (Desc_r(x))} InS(y)$.

Properties 2, 3, and 4 provide a powerful computational tool for determining which edges are possible candidate for being optimal swap edges. We will now see how to efficiently use this tool.

[†]Since the labelling of the incident links is drawn from a totally ordered set, this numbering is unique.

3.2 The Algorithm

By definition, a node x knows the weight of all its incident links, and can distinguish those that are part of T_r from those that are not; of those that are part of T_r, x can distinguish the one that leads to its parent from those leading to its children.

We assume that each node x knows its distance from r, the distances of its neighbors from r, its own pair $\alpha(x)$, as well the pairs of its neighbors. If not available, this information can be easily and efficiently acquired.

In the proposed algorithm, each node x computes an optimal swap edge for e_x, i.e., the swap edge for e_x in the shortest path from x to r in $E \setminus \{e_x\}$. We shall denote such an edge as b_x and call it the *bridge of* x. A node x also contributes, if necessary, to the computation of the bridges of other nodes.

Note that, because of Properties 2 and 3, to determine if an edge is a swap edge is sufficient to examine the relationship "descendant", which in turn is uniquely determined by the mapping α. Hence, in the following, we shall use the term "feasible for $\alpha(x)$" to mean "swap edge for e_x" without any loss of precision.

Algorithm ALL BRIDGES (reported in Algorithm 1), describes the state-event-action set of rules: it specifies what action must a node perform if in a given state.

Initially, all nodes are in state `COMPUTING` and start the execution. Each node x maintains a list $L(x)$ of possible swap edges. Initially $L(x)$ contains all the links incident on x that are not in the tree.

To each $e = (w, z) \in L(x)$, where w is a descendent of x (possibly, $x = w$), it will be associated the pair $\alpha(z)$ as well as the distance

$$d[e] = d(x, w) + |(w, z)| + d(z, r).$$

The set $L(x)$ is kept sorted w.r.t. the distances. Note that while each node knows its distance from the root, the distances $d[e]$ must be computed. This can be easily done: when a swap edge e is transmitted by a child to a parent along (u, v), together with a distance d, node v will increment the distance by $|(u, v)|$.

It is understood that when sending information about an edge e, as in the "Choice" messages, this information include the pairs of labels associated to the end nodes of e.

Nodes behaves differently depending on whether they are leafs of internal nodes in T_r.

Leaf. Leafs can compute their bridges locally: a leaf x chooses as bridge the minimum cost incident edge among those in $L(x)$, by calling routine `ChooseMin`(a, b) (Lines – in Algorithm 1).
Function `ChooseMin`(a, b) determines the swap edge with minimum distance in $L(x)$ that is feasible with (a, b). Note that for the leaves all edges in $L(x)$ are feasible. Its output is $(mybridge, d)$, where $mybridge$ is the computed edge, and d is the distance between x and r using $mybridge$ as a swap edge. If no such an edge exists, function `ChooseMin`(a, b) returns NIL.
Finally, the leaf sends to its parent a "Choice" message containing the result of `ChooseMin()`.

Internal. An internal node x waits until it receives the bridges computed by *all* its children. In particular, as soon as it receives a "Choice" message containing the pair $(edge, distance)$, it checks if $edge$ is a swap edge for itself: this is done by calling the Boolean function `Feasible(edge,` (a, b)`)`, that determines whether $edge$ is feasible with the pair of labels (a, b) (Line –); by definition, if $edge = NIL$, `Feasible` is always TRUE, regardless of (a, b). Let $e = (z, w)$. The feasibility of the swap edge e is checked by comparing the pair (a, b) with the pair corresponding to w.
If the feasibility test succeed, then the received pair is stored into array $choice$ (Line –); otherwise, x sends an explicit request (message "Request") to the child that sent the pair $(edge, distance)$, asking it to look for an edge that is feasible for (a, b) (Line –). When x receives from each of its children a feasible edge ($count = |children|$ in Line –), x can compute its bridge, and send its chosen bridge to its parent (Lines –).
When a node x receives a message ("Request", (p, q)) (i.e., x is asked to look for an edge feasible for node labelled (p, q)), it first computes the edge in $L(x)$ that is feasible for (p, q), and with minimum distance from the root (routine `ChooseMin()` in Line –). If x is a leaf, then it sends to the parent the computed edge ("Choice" message in Line –). Otherwise, for each child y of x, it first check whether the choice sent by y is feasible for (p, q) (we recall that this choice has been stored in array $choice$ of x). If the feasibility test fails, then the "Request" message is forwarded to y (i.e., y is asked to look for a feasible edge for (p, q), Line –), and x starts `WAITING` until all requested children answer (Lines –). At that time, x sends to its parent the best feasible edge for (p, q) it received from its children.
If none of the feasibility tests fail, then x does not need to forward the request to its children, and can choose as swap edge to send its parent the edge in $L(x)$ with minimum distance from the root. (i.e., the result of `ChooseMin()`, Line –).

(2) Example.

As an example consider the SPT of the graph of Figure 1, shown in Figure 2. According to the algorithm, the leaf nodes B, G, H, and E compute their bridges directly and become SWAPPED. Node F re-

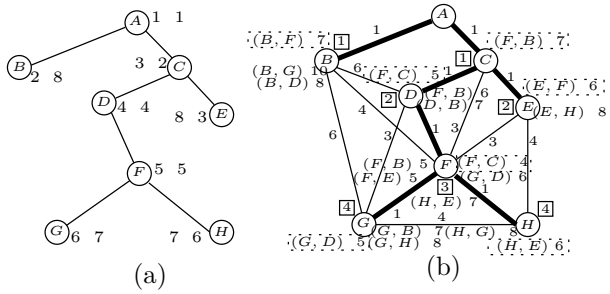

Fig. 2 Computing all the optimal swap edges in the graph G of figure 1 for the SPT rooted in A. (a) For each node x there is shown $\alpha(x)$. (b) For each node x there are shown: the distance of x from the root (in the solid box); and the feasible edges for x, together with their distances (in the dotted box).

Algorithm 1 ALL BRIDGES (G, T_r) for node x

Input: Children of x in T_r, parent of x in T_r, pair $\alpha(x) = (a, b)$ of labels associated to x in T_r, neighbors of x in G.
States: $\mathcal{S} = \{$`COMPUTING, SWAPPED, WAITING`$\}$.

```
    COMPUTING
        count := 0;
 3:     If leaf Then
          (mybridge, d) := ChooseMin(a, b);
          send ("Choice", mybridge, d) to parent;
 6:       become SWAPPED;
        If internal Then
          Receiving ("Choice",edge, distance);   \* it comes
          from a child *\
 9:       If Feasible(edge, (a, b)) Then
            count := count + 1;
            choice[sender] := (edge, distance);
12:         If count = |children| Then
              (mybridge, d) := ChooseMin(a, b);
              send ("Choice",mybridge, d) to parent;
15:           become SWAPPED;
          If Not Feasible(edge, (a, b)) Then send ("Re-
          quest", (a, b)) to sender.
    SWAPPED
18:     Receiving ("Request", (p, q));
        (edge, d) = ChooseMin(p, q);
        If leaf Then send ("Choice",edge, d) to parent .
21:     Else
          check := 0;
          For All y ∈ Children Do
24:         If Not Feasible(swap[y], (p, q)) Then
              send ("Request",(p, q)) to y;
              check := check + 1;
27:       If check > 0 Then become WAITING .
          Else
            (edge, d) := ChooseMin(p, q);
30:         send ("Choice",edge, d) to parent.
    WAITING
        Receiving ("Choice",edge, distance);
33:     choice[sender] := (edge, distance);
        check := check − 1;
        If check = 0 Then
36:       (edge, d) :=ChooseMin(p, q);
          send (edge, d) to parent;
          become SWAPPED.
```

ceives swap edges from G and H and can computes its bridge $[(F, C), 4]$ becoming SWAPPED. Node D receives the swap edge $[(F, C), 5]$ from its only child F, and this becomes its bridge. Node C instead receives non feasible edges from both D and E; it then sends to both of them a request for a feasible edge. Node E does not have edges feasible with C; hence it sends NIL. As for D, as the swap link it had received from F is not feasible for C, it will forward the request to F. Since the swap edge known to F ((F, C)) is not feasible for C, F forwards the request to the leaves G and H. At this point there is a propagation of swap edges feasible with C. In fact, G sends up $[(G, B), 7]$, H sends up NIL, F chooses as minimum $[(F, B), 5]$ and returns this information to D which sends it to C. Receiving NIL from E and $[(F, B), 6]$ from D, node C can conclude its computation selecting $[(F, B), 7]$ as its bridge.

3.3 Analysis

The correctness of Algorithm ALL BRIDGES is established by the following Theorem.

Theorem 1: In algorithm ALL BRIDGES:

(i) each node u correctly computes b_u;
(ii) if so requested by its parent, each node u will determine among the swap edges incident to its subtree and feasible with $\alpha(e) = (p, q)$, if any, one edge e' that minimizes the distance between u and r in $T_r - \{e\} \cup \{e'\}$.

Proof Removal of e_u partitions T_r in two subtrees, one rooted in r the other in u. By definition, any feasible swap edge, and hence b_u, must have an endpoint in each component. The proof will be by induction on the height $h(u)$ of the subtree of T_r rooted in u.

Basis. $h(u) = 0$; i.e., u is a leaf. In this case, one components contains only u, while the other contains all the other nodes. In other words, the only possible swap edges are incident on u. Thus, u correctly computes b_u, proving (i); it can also immediately determine the feasibility of any of those links with respect to any pair of labels, and thus answer correctly any received query, proving (ii).

Induction step. Let the theorem hold for all nodes v with $k - 1 \geq h(v) \geq 0$; we will now show that it holds for u with $h(u) = k$. Since u is not a leaf, the subtree $T_r[u]$ rooted at u contains at least two nodes. Consider the set $S(u)$ of all feasible swap edges for e_u; clearly, if $e = (w, z) \in S(u)$ then one of its end point, say w, is in $T_r[u]$ (and thus a descendent of u), while the other say z, is not.
Let v be a child of u; then $h(v) < k$. It follows that, by inductive hypothesis, when asked by u, v will send to u the edge in $Swap(v)$ that, among those feasible with e_u, minimizes the distance between v

and r. We will now show that this information is sufficient for u to correctly determine its optimal swap edge b_u.

By definition of bridge, b_u is the edge $e = (w, z)$ in $S(u)$ that minimizes the quantity $d_u[e] = d(u, w) + |(w, z)| + d(z, r)$. By Property 4, the optimal swap edge is either incident on u or on a strict descendent of u. Clearly u can locally determine its distance from r for any of its incident swap edges, and determine the minimum one. If e is not incident on u, it is in the subtree $T_r[v]$ rooted in a child v of u; furthermore, e is the swap edge in $S(v)$ that, among those feasible with e_u, minimizes the distance between v and r. In other words, once u obtains from each child v' the swap edge $e' \in S(v')$ that, among those feasible with e_u, minimizes the distance between v_i and r, u can determine the minimum one. Since, by inductive hypothesis, every child of u sends this information to u, it follows that u can correctly determine its optimal swap edge, proving Part (i) of the Theorem.

To prove Part (ii), it is sufficient to observe that, by Property 3, u can determine which of its incident swap edges are feasible with a given pair (p, q); furthermore, since the height of its children in T_r is less than k, then by inductive hypothesis it can obtain from them the "best" swap edge in their subtree feasible with (p, q). Therefore, u can determine among the swap edges incident to its subtree and feasible with (p, q), if any, one that minimizes the distance between u and r, proving Part (ii).

□

Let us now examine the message complexity of the proposed algorithm. Let n_r^* be the number of edges of the transitive closure of $T_r \setminus \{r\}$; observe that $0 \leq n^* \leq (n-1)(n-2)/2$.

Theorem 2: The message complexity of Algorithm ALL BRIDGES is at most $2n^* + n - 1$.

Proof Each node, once computed its optimal swap edge, sends a message to its parent, for a total of $n-1$ "Choice" messages. To compute its optimal swap edge, a node x might send a "Request" message to all its children (if the original information provided by them is not feasible), which in turn might send to their children (if no feasible information was received), and so on. Thus, in the worst case, each descendent of x will receive a "Request" and reply a "Choice" for a total of $2|Desc(x)|$ messages.

Since each node, except the root, must compute its optimal swap edge, this process will require at most

$$\sum_{x \neq r} 2|Desc(x)| = \sum_{x} 2(|Ance(x)| - 1) = 2n_r^*,$$

where $Ance(x)$ denotes the set of ancestors of x. □

A node, an edge, a label, a weight, and a distance are all unit of data. To evaluate the overall data complexity of the algorithm we need to consider the message size; since each message contains only a constant number of units of information, we have:

Theorem 3: The data complexity of the distributed Algorithm ALL BRIDGES is $O(n_r^*)$.

Observe that the data complexity needed by our algorithm to compute all the optimal swap edges of a shortest-path spanning-tree is no more (and very often dramatically less) than the one of computing the shortest-path spanning-tree itself [2], [3], [7].

4. An O(n) Messages Algorithm

In this section, we discuss how the algorithm of Section 3 can be modified in order to reduce the message complexity to $O(n)$ in case that longer messages are allowed. The overall information complexity of the new algorithm remains of $O(n_r^*)$.

The idea is now that each node simultaneously computes the "best" feasible swap edges, not only for itself, but also for all its ancestors in the SPT. The modified algorithm will be described only at high level. It consists simply of a *broadcast* phase started by the children of the root, followed by a *convergecast* phase started by the leaves.

Algorithm 2 ALL BRIDGES-2

[Broadcast.]

1. Each child x of the root starts the broadcast by sending a list containing $\alpha(x)$ to its children.
2. Each node y, adds $\alpha(y)$ to the received list and sends it to its children.

[Convergecast.]

1. Each leaf z first computes its own bridge. It then computes the best feasible swap edge for each of its ancestors, and sends the list of those edges to its parent (if different from r).
2. An internal node y waits until it receives the list of best swap edges from each of its children. Based on the received information and on $InS(y)$, it computes its bridge b_y. It also computes the best feasible swap edge for each of its ancestors, and sends the list of those edges to its parent (if different from r).

Theorem 4: Each node $u \neq r$:

(i) correctly computes b_u;
(ii) determines for each ancestor $v \neq r$ the best swap edge feasible with $\alpha(v)$, if any.

Proof First observe that, as a result of the broadcast, every node will receive the pair associated to each of its

ancestors (except r); hence it can determine feasibility, for each ancestor, of any available set of swap edges. The proof is by induction on the height $h(u)$ of the subtree of T_r rooted in u.

Basis. $h(u) = 0$; i.e., u is a leaf. In this case, one components contains only u, while the other contains all the other nodes. In other words, the only possible swap edges are incident on u. Thus, u correctly computes b_u, proving (i); it can also immediately determine the feasibility of any of those links with respect to any pair of labels, and thus answer correctly any received query, proving (ii).

Induction step. Let the theorem hold for all nodes x with $k - 1 \geq h(x) \geq 0$; we will now show that it holds for u with $h(u) = k$. By inductive hypothesis, it receives from each child y the best feasible swap edge for each ancestor of y, including u itself. Hence, based on these lists and on the locally available set $InSwap(u)$, u can correctly determine its optimal swap edge, as well as its best feasible swap edge for each of its ancestors.

□

The functioning of the Algorithm ALL BRIDGES-2 can be followed in the example of Figure 2.b and in particular through the convergecast, starting from the two leaves G and H. After Phase 1, G and H know their ancestors, namely: F, D, C. Node G computes its bridge as $((G, D), 5)$, and the minimum swap edge for each of its ancestor, namely: $((G, D), 5)$ for F, $((G, D), 5)$ for D and $((G, B), 7)$ for C, and sends these values to F. Similarly H computes its bridge as $((H, E), 6)$, and the minimum swap edge for each ancestor, namely: $((H, E), 6)$ for F, $((H, E), 6)$ for D and NIL for C, and sends these values to F. F computes its optimal swap edge as the minimum among its incident edges, that is, $((F, C), 4)$, the edge coming from G, (G, D), having now distance 6, and the edge coming from H, (H, E), having now distance 7. Hence it selects $((F, C), 4)$, as bridge and computes the minimum feasible swap edge for each of its ancestor, namely: $((F, C), 4)$ for D, and $((F, B), 5)$ for C, and sends these values to F. D selects as bridge the edge coming from F, with distance incremented by 1, that is $((F, C), 5)$, and sends $((F, B), 6)$ for C. C can finally selects its bridge, considering the information coming from D and that coming from E which is NIL, as $((F, B), 7)$.

Let us now analyze the complexity of the algorithm:

Theorem 5:
The message complexity of Algorithm ALL BRIDGES-2 is exactly $2(n - 1 - \delta(r))$, where $\delta(r)$ is the degree of r.

Proof In the broadcast phase, every node except the root and its children receives a message. In the convergecast phase, every node except the root and its children sends a message. □

Theorem 6: The data complexity of Algorithm ALL BRIDGES-2 is exactly $2n^*_r$.

Proof In the broadcast phase, every node (except the root and its children) receives the labels of all its ancestors. In the convergecast phase, every node (except the root and its children) sends a swap edge for each of its ancestors. □

5. Point-of-failure Shortest-path Re-routing

5.1 Computing for all Destinations

We have seen how to efficiently compute the optimal swap edges for a single SPT of a biconnected graph. To construct fault tolerant shortest-path persistent routing tables, the proposed computation must be carried out for all the n shortest path trees, each having as root a different vertex of the graph. The computation of the resulting costs is rather straightforward. For example, using long messages we have

Theorem 7: All optimal swap edges for all n SPTs T_r can be constructed with $2n^2 - 4m - 2n$ messages, where m is the number of edges; the overall information complexity is less than n^3.

Proof We execute Algorithm ALL BRIDGES-2 n times, each time with a different node as the root. Then, by Theorem 5, it follows that the total number of messages will be

$$\sum_r 2(n - 1 - \delta(r)) = 2n(n - 1) - 2 \sum_r \delta(r) = 2n(n - 1) - 4m.$$

For each T_r, we have that $0 \leq n^* \leq (n - 1)(n - 2)/2$; hence, by Theorem 6, it follows that the overall information complexity will be less than $n(n - 1)(n - 2)$. □

Notice that this cost is always less, and often significantly so, than that required to construct the all-pairs shortest paths using long messages

5.2 Routing with optimal swap edges

Let us now address the problem of how the routing tables are organized and how the information stored there must be used in presence of failure of an incident link.

The routing table at node u contains, for each destination r, the neighbor v in the shortest path from u to r (as determined in the SPT T_r) as well as the optimal swap edge (u', v') for (u, v) in T_r. This entry will be

indicated as $RT[u, r].bridge$ in the following.

Consider now a message M with destination r arriving to u where however the link to the next hop v has just failed. The following steps are performed in this case:

1. the optimal swap edge (u', v') for (u, v) is retrieved by u from $RT[u, r].bridge$;
2. the message M is backtracked to u' along the shortest-path from u to u';
3. M is transmitted by u' along the bridge (u', v');
4. M is sent by v' to the final destination r using the information in its routing table.

Note that the optimal swap edge can be incident to u; in this case u and u' coincide and there is no backtracking. Otherwise, the backtracking is accomplished according to the standard routing table information, but with final destination u' instead of r. This guarantees that M is sent on the shortest path between u and u'.

Hence, each message M can travel in two different modes:

- the *normal* mode, in which it follows the standard shortest-path routing algorithm to its final destination, and
- the *backtracking* mode, in which it follows the standard shortest-path routing algorithm to a temporary destination (from u to u' in Step 2 above) and crosses a optimal swap edge (edge (u', v') in Step 3 above).

Clearly each message must contain one additional bit to specify the routing mode (0 for normal, 1 for backtracking). Furthermore, when backtracking, the information about the bridge (u', v') must be added to the message; notice that this information includes the temporary destination u'. In other words, the format of a message is as following

- < 0, *destination*, $M >$ for the normal mode
- < 1, *bridge*, *destination*, $M >$ for the backtracking mode

When a node u recognizes a failure along one of its edges, it reads in the routing table the bridge (u', v') for the failed edge; then, in case u' is different from u, it modifies the label of the messages that need to transit along the failed edge, and send them to the next hop for destination u'. Otherwise the messages are sent directly to v' without changing the mode. More precisely, the routing algorithm is reported in Algorithm 3.

Referring to Figure 2.b, in presence of a failure on the edge (C, A) a message in C with destination A is backtracked to F, it traverses the bridge (F, B) and then continues to its destination in normal mode. Note that in the algorithm, node u cannot detect a failure when it is routing a message in swap mode, because the model is 1-fault tolerant only.

As concluding remarks of this section, note that more than one optimal swap edge can be incident on the same vertex. For instance, in Figure 2.b, the optimal swap edge for (D, C) is (F, C), and the optimal swap edge for (C, A) is (F, B), both incident on F. Then, storing only one endpoint of the bridge is not sufficient. For this reason, the additional information we use in the routing tables, of size $2l$ to store one swap edge for each failure, is minimal. Other solutions (see for example [10]) storing only one vertex for each failure, that is with additional information of size l, are not able to guarantee the shortest alternative path.

Algorithm 3 ROUTING for Node u

```
Receiving (0, dest, M);
l := RT[u, dest];
If l = dest Then
  Process M.  \* Reached Destination *\
Else
  If There is a failure in (u, l) Then
    (x, y) := RT[u, dest].bridge;
    If u ≠ x Then
      send (1, (x, y), dest, M) to RT[u, x];
    Else
      send (0, dest, M) to y;
  Else  \* There is not a failure in (u, l) *\
    send (0, dest, M) to l;

Receiving (1, (x, y), dest, M)
l = RT[u, x];
If u ≠ x Then
  send (1, (x, y), dest, M) to l;
Else
  send (0, dest, M) to y;
```

6. Concluding Remarks

In this paper we have presented simple and efficient distributed algorithms for computing the optimal swap edges of a shortest-path tree. One algorithm uses messages containing a constant amount of information, while the other is tailored for systems that allow long messages. Both algorithms exchange a quantity of information which is no more, and often significantly so, than that required to construct the shortest-path spanning-tree; also, they require information that can be acquired, with no increase in order of magnitude, during the shortest-path spanning-tree construction.

The idea of point-of-failure swap rerouting can be applied to trees other than shortest-path trees (e.g., [4], [13]). In this case, different criteria of optimality must be employed. We are currently investigating whether our technique could be generalized and extended to those other situations.

The proposed algorithms allow for the efficient construction of point-of-failure shortest-path rerouting service. To do so, the proposed computation must be carried out for the n shortest path trees, each having as root a different vertex of the graph. In this regards,

an interesting open problem is whether it is possible to achieve the same goal in a more efficient way than by performing n independent computations. For example, it is known that the constructions of all-pairs shortest-paths can be done more efficiently than n independent constructions of a single shortest-path spanning-tree (e.g., [1]); the research question is whether something similar holds also in this context.

Among the possible development of this study, it would be interesting to study how to recover from multiple link failures, following the same strategy of storing in the routing tables the information useful for finding alternative paths. The problem appears much more complex. In addition, the size of the routing tables is limited [15], hence the additional information to store in order to compute alternative paths in presence of faults must be also be small. An interesting problem is to determine the minimal amount of memory needed to obtain a k-fault-tolerant point-of-failure shortest-path rerouting service.

(3) Acknowledgments.

Part of this work was carried out while the the first and last authors were visited by the others at the University of Ottawa and at Carleton University. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, and by "Progetto ALINWEB: Algoritmica per Internet e per il Web", MIUR Programmi di Ricerca Scientifica di Rilevante Interesse Nazionale.

References

[1] Y. Afek and M. Ricklin. Sparser: a paradigm for running distributed algorithms. *Journal of Algorithms*, 14:316-328, 1993.

[2] B. Awerbuch and R. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, 33:315–322, 1987.

[3] K. M. Chandy and J. Misra. Distributed computation on graphs: shortest path algorithms. *Communication of ACM*, 25:833–837, 1982.

[4] A. Di Salvo and G. Proietti. Swapping a failing edge of a shortest paths tree by minimizing the average stretch factor. *Proc. of 10th Colloquium on Structural Information and Communication Complexity* (SIROCCO 2004) 2004.

[5] D. Eppstein, Z. Galil, and G.F. Italiano./ Dynamic graph algorithms. *CRC Handbook of Algorithms and Theory, CRC Press, 1997.*

[6] P. Flocchini, T. Mesa, L. Pagli, G. Prencipe, and N. Santoro. Efficient protocols for computing optimal swap edges. *In Proc. of 3rd IFIP International Conference on Theoretical Computer Science* (TCS 2004), 2004, to appear.

[7] G. N. Frederikson. A distributed shortest path algorithm for planar networks. *Information and Computation*, 86:140-159, 1990.

[8] P. Humblet. Another adaptive distributed shortest path algorithm. *IEEE/ACM Transactions on Communications*, 39(6):995–1003, 1991.

[9] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. *Information and Computation*, 79:43-59, 1988.

[10] H. Ito, K. Iwama, Y. Okabe and T. Yoshihiro. Polynomial-time computable backup tables for shortest-path routing. *Proc. of 10th Colloquium on Structural Information and Communication Complexity* (SIROCCO 2003), 163–177, 2003.

[11] H. Mohanty and G.P.Bhattacharjee. A distributed algorithm for edge-disjoint path problem *Proc. of 6th Conference on Foundations of Software Technology and Theoretical Computer Science* (FSTTCS), 44-361, 1986.

[12] P. Narvaez, K.Y. Siu, and H.Y. Teng. New dynamic algorithms for shortest path tree computation *IEEE Transactions on Networking*, 8:735–746, 2000.

[13] E. Nardelli, G. Proietti, and P. Widmayer. Finding all the best swaps of a minimum diameter spanning tree under transient edge failures. *Journal of Graph Algorithms and Applications*, 2(1):1–23, 1997.

[14] E. Nardelli, G. Proietti, and P. Widmayer. Swapping a failing edge of a single source shortest paths tree is good and fast. *Algoritmica*, 35:56–74, 2003.

[15] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach, 3rd Edition*. Morgan Kaufmann, 2003.

[16] G. Proietti. Dynamic maintenance versus swapping: An experimental study on shortest paths trees. *Proc. 3rd Workshop on Algorithm Engineering* (WAE 2000), 207–217, 2000.

[17] R. E.Tarjan. Application of path compression on balanced trees. *Journal of ACM*, 26:690–715, 1979.