A Distributed Algorithm for Finding All Best Swap Edges of a Minimum Diameter Spanning Tree*

Beat Gfeller[1], Nicola Santoro[2], and Peter Widmayer[1]

[1] Institute of Theoretical Computer Science, ETH Zurich, Switzerland
{gfeller,widmayer}@inf.ethz.ch
[2] School of Computer Science, Carleton University, Ottawa, Canada
santoro@scs.carleton.ca

Abstract. Communication in networks suffers if a link fails. When the links are edges of a tree that has been chosen from an underlying graph of all possible links, a broken link even disconnects the network. Most often, the link is restored rapidly. A good policy to deal with this sort of *transient* link failures is *swap rerouting*, where the temporarily broken link is replaced by a single *swap* link from the underlying graph. A rapid replacement of a broken link by a swap link is only possible if all swap links have been precomputed. The selection of high quality swap links is essential; it must follow the same objective as the originally chosen communication subnetwork. We are interested in a minimum diameter tree in a graph with edge weights (so as to minimize the maximum travel time of messages). Hence, each swap link must minimize (among all possible swaps) the diameter of the tree that results from swapping. We propose a distributed algorithm that efficiently computes all of these swap links, and we explain how to route messages across swap edges with a compact routing scheme.

1 Introduction

For communication in computer networks, often only a subset of the available connections is used to communicate at any given time. If all nodes are connected using the smallest number of links, the subset forms a spanning tree of the network. Depending on the purpose of the network, there is a variety of desirable properties of a spanning tree. We are interested in a *Minimum Diameter Spanning Tree* (MDST), i.e., a tree that minimizes the largest distance between any pair of nodes, thus minimizing the worst case length of any transmission path. The importance of minimizing the diameter of a spanning tree has been widely recognized (see e.g. [2]); essentially, the diameter of a network provides a lower bound on the computation time of most algorithms in which all nodes participate.

* We gratefully acknowledge the support of the Swiss SBF under contract no. C05.0047 within COST-295 (DYNAMO) of the European Union and the support of the Natural Sciences and Engineering Research Council of Canada.

A. Pelc (Ed.): DISC 2007, LNCS 4731, pp. 268–282, 2007.

One downside of using a spanning tree is that a single link failure disconnects the network. Whenever the link failure is transient, i.e., the failed link soon becomes operational again, the best possible way of reconnecting the network is to replace the failed link by a single other link, called a *swap* link. Among all possible swap links, one should choose a *best swap* w.r.t. the original objective [5,6,7,8], that is in our case, a swap that minimizes the diameter of the resulting *swap tree*. Note that the swap tree is different from a minimum diameter spanning tree of the underlying graph that does not use the failed link. The reason for preferring the swap tree to the latter lies in the effort that a change of the current communication tree requires: If we were to replace the original MDST by a tree whose edge set can be very different, we would need to put many edges out of service, many new edges into service, and adjust many routing tables substantially — and all of this for a transient situation. For a swap tree, instead, only one new edge goes into service, and routing can be adjusted with little effort (as we will show). Interestingly, this choice of swapping against adjusting an entire tree even comes at a moderate loss in diameter: The swap tree diameter is at most a factor of 2.5 larger than the diameter of an entirely adjusted tree [6].

In order to keep the required time for swapping small, for each edge of the tree, a best swap edge is precomputed. We show in the following that this distributed computation of *all best swaps* has the further advantage of gaining efficiency (against computing swap edges individually), because dependencies between the computations for different failing edges can be exploited.

Related Work. Nardelli et al. [6] describe a centralized (i.e., non-distributed) algorithm for computing all best swaps of a MDST in $O(n\sqrt{m})$ time and $O(m)$ space, where the given underlying communication network $G = (V, E)$ has $n = |V|$ vertices and $m = |E|$ edges. For shortest paths trees, an earlier centralized algorithm has been complemented by a distributed algorithm [7] using totally different techniques for finding all best swap edges for several objectives [3,4], with either $O(n)$ messages of size $O(n)$ (i.e., a message contains $O(n)$ node labels, edge weights, etc.) each, or $O(n^*)$ short messages with size $O(1)$ each, where n^* denotes the size of the transitive closure of the tree, where edges are directed away from the root. In a so-called preprocessing phase of this algorithm, some information is computed along with the spanning tree construction using $O(m)$ messages. A distributed algorithm for computing a MDST in a graph $G(V, E)$ in an asynchronous setting has $O(n)$ time complexity (in the standard sense, as explained in Section 3) and uses $O(nm)$ messages [2]. However, no efficient distributed algorithm to compute the best swaps of a MDST had been found to date.

Our Contribution. In this paper, we propose a distributed algorithm for computing all best swaps of a MDST using no more than $O(\max\{n^*, m\})$ messages of size $O(1)$ each. The *size of a message* denotes the number of atomic values that it contains, such as node labels, edge weights, path lengths etc., and n^* is the size of the transitive closure of the MDST with edges directed away from a center of the tree. Both n^* and m are very natural bounds: When each subtree triggers as many messages as there are nodes in the subtree, the size of the transitive closure describes the total number of messages. Furthermore, it seems inevitable that each node receives some information from each of its neighbours in G, across each potential swap edge. Our algorithm runs in $O(\|\mathcal{D}\|)$ time (in the standard sense, as explained in Section 3), where $\|\mathcal{D}\|$ is the hop-length of the diameter path of G; note that this is asymptotically optimal. The message and time costs of our algorithm are easily subsumed by the costs of constructing a MDST distributively using the algorithm from [2]. Thus, it is cheap to precompute all the best swaps in addition to constructing a MDST initially.

Just like the best swaps algorithms for shortest paths trees [3,4], our algorithm (like many fundamental distributed algorithms) exploits the structure of the tree. This tree, however, is substantially different in that it requires a significantly more complex invariant to be maintained during the computation: We need to have just the right collection of pieces of paths available so that on the one hand, these pieces can be maintained efficiently, and on the other hand, they can be composed to reveal the diameter at the corresponding steps in the computation.

Furthermore, we propose a compact routing scheme for trees which can quickly and inexpensively adapt routing when a failing edge is replaced by a best swap edge. Notably, our scheme does not require an additional full backup table, but assigns a label of $c \log n$ bits to each node (for some small constant c); a node of degree δ stores the labels of all its neighbours (and itself), which amounts to $\delta c \log n$ bits per node, or $mc \log n$ bits in total. Given this labelling, knowledge of the labels of both adjacent nodes of a failing edge and the labels of both adjacent nodes of its swap edge is sufficient to adjust routing.

In Section 2, we formally define the *distributed all best swaps* problem. Section 3 states our assumptions about the distributed setting and explains the basic idea of our algorithm. In Section 4, we study the structure of diameter paths after swapping, and we propose an algorithm for finding best swaps. The algorithm uses information that is computed in a preprocessing phase, described in Section 5. Our routing scheme is presented in Section 6. Section 7 concludes the paper.

2 Problem Statement and Terminology

A communication network is a 2-connected, undirected, edge weighted graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges. Each edge $e \in E$ has a non-negative real *length* $w(e)$. The length $|\mathcal{P}|$ of a path $\mathcal{P}$ is the sum of the lengths of its edges, and the *distance* $d(x, y)$ between two vertices x, y is the length of a shortest path between x and y. The *hop-length* $\|\mathcal{P}\|$ of a path $\mathcal{P}$ is the number of edges that $\mathcal{P}$ contains. Throughout the paper, we are only dealing with *simple* paths. Given a spanning tree $T = (V, E(T))$ of G, let $\mathcal{D}(T) := \langle d_1, d_2, \ldots, d_k \rangle$ denote a *diameter* of T, that is, a longest path in T (see Fig. 1). Where no confusion arises, we abbreviate $\mathcal{D}(T)$ with $\mathcal{D}$. Furthermore, define the *center* d_c of $\mathcal{D}$ as a node such that the lengths of $\mathcal{D}_L := \langle d_1, d_2, \ldots, d_c \rangle$ and $\mathcal{D}_R := \langle d_c, d_{c+1}, \ldots, d_k \rangle$ satisfy $|\mathcal{D}_L| \geq |\mathcal{D}_R|$ and have the smallest possible difference $|\mathcal{D}_L| - |\mathcal{D}_R|$. The set of neighbours of a node z (excluding z itself) in G and in T is written as $N_G(z)$ and $N_T(z) \subseteq N_G(z)$, respectively.

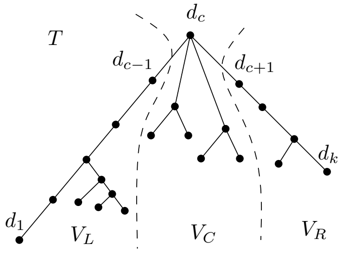

Fig. 1. A minimum diameter spanning tree T

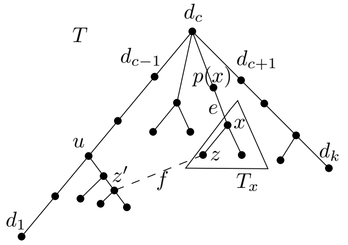

Fig. 2. A swap edge $f = (z, z')$ for $e = (x, p(x))$

Let T be rooted at d_c, and let, for each node $x \neq d_c$, node $p(x)$ be the *parent* of x and $C(x)$ the set of its *children*. Furthermore, let $T_x = (V(T_x), E(T_x))$ be the subtree of T rooted at x, including x. Let V_L (L stands for “left”) be the set of nodes in the subtree rooted at d_{c-1}, V_R the set of nodes in the subtree rooted at d_{c+1}, and V_C all other nodes.

Now, the removal of any edge $e = (x, p(x))$ of T partitions the spanning tree into two trees T_x and $T \backslash T_x$ (see Fig. 2). Note that $T \backslash T_x$ does not contain the node x. A *swap edge* f for e is any edge in $E \backslash E(T)$ that (re-)connects T_x and $T \backslash T_x$, i.e., for which $T \backslash \{e\} \cup \{f\} =: T_{e/f}$ is a spanning tree of $G \backslash \{e\}$. Let $S(e)$ be the set of swap edges for e. A *best swap edge* for e is any edge $f \in S(e)$ for which $|\mathcal{D}(T_{e/f})|$ is minimum. A *local* swap edge of node z for some failing edge e is an edge in $S(e)$ adjacent to z. The *distributed all best swaps* problem for a MDST is the problem of finding for every edge $e \in E(T)$ a best swap edge (with respect to the diameter). Throughout the paper, let $e = (x, p(x))$ denote a failing edge and $f = (z, z')$ a swap edge, where z is a node inside T_x, and z' a node in $T \backslash T_x$.

3 Algorithmic Setting and Basic Idea

In our setting, nodes have unique identifiers that possess a linear order. Further, let each node know its own neighbours in T and in G, and for each neighbour the length of the corresponding edge. We assume port-to-port communication between neighbouring nodes. The distributed system of nodes is totally asynchronous. Each message sent from some node to one of its neighbours eventually arrives (there is no message loss). As usual, we define the asynchronous time complexity of an algorithm as the longest possible execution time assuming that sending a message requires at most one time unit. Furthermore, nodes do not need to know the total number of nodes in the system (although it is easy to count the nodes in T using a convergecast).

3.1 The Basic Idea

Our goal is to compute, for each edge of T, a best swap edge. A swap edge for a given failing edge $e = (x, p(x))$ must connect the subtree of T rooted at x to the part of the tree containing $p(x)$. Thus, a swap edge must be adjacent to some node inside T_x. If each node in T_x considers its own local swap edges for e, then in total all swap edges for e are considered. Therefore, each node inside T_x finds a best *local* swap edge, and then participates in a *minimum finding process* that computes a (globally) best swap edge for e. The computation of the best local swap edges is composed of three main phases: In a first preprocessing phase, a root of the MDST is chosen, and various pieces of information (explained later) are computed for each node. Then, in a top-down phase each node computes and forwards some "enabling information" (explained later) for each node in its own subtree. This information is collected and merged in a third bottom-up phase, during which each node obtains its best local swap edge for each (potentially failing) edge on its path to the root. The efficiency of our algorithm will be due to our careful choice of the various pieces of information that we collect and use in these phases.

To give an overview, we now briefly sketch how each node computes a best local swap edge. First observe that after replacing edge e by f, the resulting diameter is longer than the previous diameter only if there is a path through f which is longer than the previous diameter, in which case the path through f is the new diameter. In this case, the length of the diameter equals the length of a longest path through f in the new tree. For a local swap edge $f = (z, z')$ connecting node $z \in V(T_x)$ and $z' \in V \backslash V(T_x)$, such a path consists of

(i) a longest path inside $T \backslash T_x$ starting in z',
(ii) edge f, and
(iii) a longest path inside T_x starting in z.

Part (i) is computed in a preprocessing phase, as described in Section 5. Part (ii) is by assumption known to z, because f is adjacent to z. Part (iii) is inductively computed by a process starting from the root x of T_x, and stopping in the leaves, as follows. A path starting in z and staying inside T_x either descends to a child of z (if any), or goes up to $p(z)$ (if $p(z)$ is still in T_x) and continues within $T_x \backslash T_z$. For the special case where $z = x$, node x needs to consider only the heights of the subtrees rooted at its children. All other nodes z in T_x additionally need to know the length of a longest path starting at $p(z)$ and staying inside $T_x \backslash T_z$. This additional *enabling information* will be computed by $p(z)$ and then be sent to z.

Once the best local swap edges are known, a best (global) swap edge is identified by a single minimum finding process that starts at the leaves of T_x and ends in node x. To compute all best swap edges of T, this procedure is executed separately for each edge of T. This approach will turn out to work with the desired efficiency:

Main Theorem. *All best swap edges of a MDST can be computed in an asynchronous distributed setting with* $O(\max\{n^*, m\})$ *messages of constant size, and in* $O(\|\mathcal{D}\|)$ *time.*

We will prove this theorem in the next sections, by proving that the preprocessing phase can be realized with $O(m)$ messages, and after that the computation of all best swap edges requires at most $O(n^*)$ additional messages.

This algorithm requires that each node knows which of its neighbours are children and which neighbour is its parent in T. Although this information is not known a priori, it can be easily computed in a preprocessing phase, during which a particular diameter and a root of T are chosen.

4 How to Pick a Best Swap Edge

In our distributed algorithm, we compute for each (potentially) failing edge the resulting new diameter for each possible swap edge candidate. This approach can be made efficient by exploiting the structure of diameter path changes, as described in the following.

4.1 The Structure of Diameter Path Changes

For a given failing edge e, let $\mathcal{P}_f$ be a longest path in $T_{e/f}$ that goes through swap edge f for e. Then, we have the following:

Lemma 1. *The length of the diameter of* $T_{e/f}$ *is* $|\mathcal{D}(T_{e/f})| = \max\{|\mathcal{D}(T)|, |\mathcal{P}_f|\}$.

Proof. Let T_1 and T_2 be the parts into which T is split if e is removed. It is easy to see that

$$|\mathcal{D}(T_{e/f})| = \max\{|\mathcal{D}(T_1)|, |\mathcal{D}(T_2)|, |\mathcal{P}_f|\}. \tag{1}$$

Since T is a MDST, we have

$$|\mathcal{D}(T_{e/f})| \geq |\mathcal{D}(T)|. \tag{2}$$

Because T_1 and T_2 are contained in T,

$$|\mathcal{D}(T_1)| \leq |\mathcal{D}(T)| \quad \text{and} \quad |\mathcal{D}(T_2)| \leq |\mathcal{D}(T)|. \tag{3}$$

If $|\mathcal{P}_f| \geq |\mathcal{D}(T)|$, it is clear that $|\mathcal{P}_f|$ is a largest term in (1), so the claim holds. On the other hand, if $|\mathcal{P}_f| < |\mathcal{D}(T)|$, then either T_1 or T_2 must contain a diameter of length exactly $|\mathcal{D}(T)|$ (otherwise, either (2) or (3) would be violated). Thus, the claim holds also in this case. □

That is, for computing the resulting diameter length for a given swap edge $f = (z, z')$ for e, we only need to compute the length of a longest path in $T_{e/f}$ that goes through f. For node z in the subtree T_x of T rooted in x, and z' outside this subtree, such a path $\mathcal{P}_f$ consists of three parts. To describe these parts, let $\mathcal{L}(H, r)$ denote a longest path starting in node r and staying inside the graph H. The first part is a longest path $\mathcal{L}(T \backslash T_x, z')$ in $T \backslash T_x$ that starts in z'. The second part is the edge f itself. The third part is a longest path $\mathcal{L}(T_x, z)$ starting in z and staying inside T_x. This determines the length of a longest path through f as $|\mathcal{P}_f| = |\mathcal{L}(T_x, z)| + w(f) + |\mathcal{L}(T \backslash T_x, z')|$.

4.2 Distributed Computation of $|\mathcal{L}(T_x, z)|$

For a given failing edge $e = (x, p(x))$, each node z in T_x needs its $|\mathcal{L}(T_x, z)|$ value to check for the new diameter when using a swap edge. This is achieved by a distributed computation, starting in x. As x knows the heights of the subtrees of all its children (from the preprocessing), it can locally compute the height of its own subtree T_x as $|\mathcal{L}(T_x, x)| = \max_{q \in C(x)}\{w(x, q) + \mathit{height}(T_q)\}$, where $C(x)$ is the set of children of x. For a node z in the subtree rooted at x, a longest simple path either goes from z to its parent and hence has length $|\mathcal{L}(T_x \backslash T_z \cup \{z\}, z)|$, or goes into the subtree of one of its children and hence has length $|\mathcal{L}(T_z, z)|$ (see Fig. 3). The latter term has just been described, and the former can be computed by induction by the parent r of z and can be sent to z. This inductive step is identical to the step just described, except that z itself is no candidate subtree for a path starting at r in the induction. In total, each node r computes, for each of its children $q \in C(r)$, the value of

$$|\mathcal{L}(T_x \backslash T_q \cup \{q\}, q)| = \\ w(q, r) + \max\left\{|\mathcal{L}(T_x \backslash T_r \cup \{r\}, r)|, \max_{s \in C(r), s \neq q}\{w(r, s) + \mathit{height}(T_s)\}\right\},$$

and sends it to q, where we assume that the value $|\mathcal{L}(T_x \backslash T_r \cup \{r\}, r)|$ was previously sent to r by $p(r)$.

A bird's eye view of the process shows that each node z first computes $|\mathcal{L}(T_x, z)|$, and then computes and sends $|\mathcal{L}(T_x \backslash T_q \cup \{q\}, q)|$ for each of its children $q \in C(z)$. Computation of the $|\mathcal{L}(T_x, z)|$ values finishes in T_x's leaves. Note that a second value will be added to the enabling information if $(x, p(x)) \in \mathcal{D}$, for reasons explained in the next section.

4.3 Distributed Computation of $|\mathcal{L}(T \backslash T_x, z')|$

In the following, we explain how z can compute $|\mathcal{L}(T \backslash T_x, z')|$ for a given swap edge $f = (z, z')$. In case the failing edge $e = (x, p(x)) \notin \mathcal{D}$, we show below that the information obtained in the preprocessing phase is sufficient.

For the sake of clarity, we analyze two cases separately, starting with the simpler case.

Case 1: The removed edge e is not on the diameter. For this case, we know from [6] that at least one of the longest paths in $T \backslash T_x$ starting from z' contains d_c. If $z' \in V_L$, we get a longest path from z' through d_c by continuing on the diameter up to d_k, and hence we have $|\mathcal{L}(T \backslash T_x, z')| = d(z', d_c) + |\mathcal{D}_R|$. If z' is in V_C or V_R, some longest path from z' through d_c continues on the diameter up to d_1, yielding $|\mathcal{L}(T \backslash T_x, z')| = d(z', d_c) + |\mathcal{D}_L|$. Remarkably, in this case $|\mathcal{L}(T \backslash T_x, z')|$ does not depend on the concrete failing edge $e = (x, p(x))$, apart from the fact that (z, z') must be a swap edge for e.

Case 2: The removed edge e is on the diameter. We analyze the case $e \in \mathcal{D}_L$, and omit the symmetric case $e \in \mathcal{D}_R$. If $z' \in V_L$ or $z' \in V_C$, we know from [6] that again, one of the longest paths in $T \backslash T_x$ starting at z' contains d_c.

Thus, for $z' \in V_L$ we are in the same situation as for the failing edge not on the diameter, leading to $|\mathcal{L}(T\backslash T_x, z')| = d(z', d_c) + |\mathcal{D}_R|$. For $z' \in V_C$, after d_c a longest path may continue either on $\mathcal{D}_R$, or continue to nodes in V_L. In the latter case, the path now cannot continue on $\mathcal{D}_L$ until it reaches d_1, because edge e lies on $\mathcal{D}_L$. Instead, we are interested in the length of a longest path that starts at d_c, proceeds into V_L, but does not go below the parent $p(x)$ of x on $\mathcal{D}_L$; let us call this length $\lambda(p(x))$. As announced before, we include the $\lambda(p(x))$ value as a second value into the *enabling information* received by $p(x)$; then, we get $|\mathcal{L}(T\backslash T_x, z')| = d(z', d_c) + \max\{|\mathcal{D}_R|, \lambda(p(x))\}$. It remains to consider $z' \in V_R$. For this case (see Fig. 4), we know (from [6]) that at least one of the longest paths in $T\backslash T_x$ starting at z' passes through the node u' closest to z' on $\mathcal{D}(T)$. After u', this path may either continue on $\mathcal{D}_R$ up to d_k, or continue through d_c going inside V_C or V_L (without crossing $e = (x, p(x))$), or continue towards d_c only up to some node d_i on $\mathcal{D}_r$, going further on non-diameter edges inside V_R. It remains to show how the length of a longest path of this last type can be found efficiently. We propose to combine three lengths, in addition to the length of the path from z' to u'. The first is the length of a longest path inside V_R that starts at d_k; let us call this length μ_R. In general, this path goes up the diameter path $\mathcal{D}_R$ for a while, and then turns down into a subtree of V_R, away from the diameter, at a diameter node that we call ρ_R (see Fig. 4). Given μ_R, the distance from u' to ρ_R, and the distance from ρ_R to d_k, the desired path length of an upwards turning path inside V_R is $d(z', u') + d(u', \rho_R) + \mu_R - d(d_k, \rho_R)$. Note that while it may seem that ρ_R needs to lie above u' on $\mathcal{D}_R$, this is not really needed in our computation, because the term above will not be largest (among all path choices) if ρ_R happens to be below or at u'. In total, we get $|\mathcal{L}(T\backslash T_x, z')| = \max\left\{d(z', d_k), d(z', d_c) + \lambda(p(x)),\ \ d(z', u') + d\big(u', \rho_R\big) + \mu_R - d\big(d_k, \rho_R\big)\right\}$.[1]

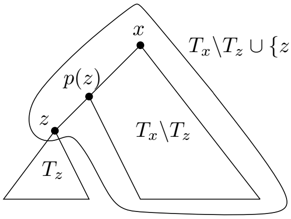

Fig. 3. Illustration of the tree $T_x\backslash T_z \cup \{z\}$

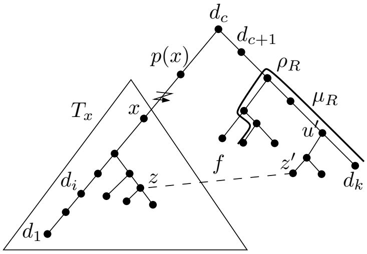

Fig. 4. Computing $|\mathcal{L}(T\backslash T_x, z')|$ if $e \in \mathcal{D}_L$, $z \in V_L$ and $z' \in V_R$

All of these path length computations can be carried out locally with no message exchanges, if the constituents of these sums are available locally at a node. We will show in the next section how to achieve this in an efficient preprocessing phase.

[1] Recall that in the definition of $\lambda(p(x))$, paths inside V_C starting from d_c are also considered.

4.4 The BestDiamSwap Algorithm

For a given edge $e = (x, p(x))$ that may fail, each node z in the subtree T_x rooted at x executes the following steps:

(i) Wait for the enabling information from the parent (unless $x = z$), and then compute $|\mathcal{L}(T_x, z)|$. Compute the enabling information for all children and send it.

(ii) For each local swap edge $f = (z, z')$, compute $|\mathcal{L}(T \backslash T_x, z')|$ as described in Section 4.3.

(iii) For each local swap edge $f = (z, z')$, locally compute

$$|\mathcal{D}(T_{e/f})| = \max\{|\mathcal{D}(T)|, |\mathcal{L}(T_x, z)| + w(f) + |\mathcal{L}(T \backslash T_x, z')|\}.$$

Among these, choose a best swap edge f^*_{local} and store the resulting new diameter as $|\mathcal{D}(T_{e/f^*_{local}})|$.

(iv) From each child $q \in C(z)$, receive the node label of a best swap edge candidate f^*_q and its resulting diameter $|\mathcal{D}(T_{e/f^*_q})|$. Pick a best swap edge candidate f^*_b among these, i.e., choose $b := \arg\min_{q \in C(z)} |\mathcal{D}(T_{e/f^*_q})|$. Compare the resulting diameter of f^*_b and f^*_{local}, and define f_{best} as the edge achieving the smaller diameter (or any of them if their length is equal), and its diameter as $|\mathcal{D}(T_{e/f_{best}})|$.

(v) Send the information f_{best}, $|\mathcal{D}(T_{e/f_{best}})|$ to the parent.

The above algorithm computes the best swap edge for one (potentially) failing edge e, based on the information available after the preprocessing phase. In order to compute all best swap edges of T, we execute this algorithm for each edge of T independently.

Analysis of the Algorithm. We now show that the proposed algorithm indeed meets our efficiency requirements:

Theorem 1. *After preprocessing, executing the* BestDiamSwap *algorithm independently for each and every edge $e \in E(T)$ costs at most $O(n^*)$ messages of size $O(1)$ each, and $O(\|\mathcal{D}\|)$ time, using a "Farthest-to-Go" queuing policy [1].*

Proof. Correctness follows from the preceding discussion. Preprocessing ensures that all precomputed values (such as $|\mathcal{L}(T \backslash T_x, z')|$) defined for the other end z' of a candidate swap edge are available locally at z'. As to the message complexity, consider the execution of the BestDiamSwap algorithm for one particular edge $e = (x, p(x))$. Starting in node $x \in V \backslash \{d_c\}$, each node in T_x sends a message containing the "enabling information" (i.e., $\mathcal{L}(T_x \backslash T_q, q)$ and possibly $\lambda(p(x))$) containing $O(1)$ items to each of its children. Furthermore, each node in T_x (including finally x) sends another message with size $O(1)$ up to its parent in the minimum finding process. Hence, two messages of size $O(1)$ are sent across each edge of T_x, and one message is sent across e. Thus, the computation of a best swap for e requires $2 \cdot |E(T_x)| + 1 = 2 \cdot |V(T_x)| - 1$ messages. The number of messages exchanged for computing a best swap edge for *each and every* edge $(x, p(x))$ where $x \in V \backslash \{d_c\}$ is $\sum_x (2 \cdot |V(T_x)| - 1) = 2n^* - (n - 1)$.

As to the time complexity, note that the best swap computation of a *single* edge according to the BestDiamSwap algorithm requires at most $O(\|\mathcal{D}\|)$ time. Now note that this algorithm can be executed independently (and thus concurrently) for each potential failing edge: In this fashion, each node x in T sends exactly one message to each node in T_x during the top-down phase. Symmetrically, in the bottom-up phase, each node u in T sends exactly one message to each node on its path to the root. The crucial point here is to avoid that some of these messages block others for some time (as only one message can traverse a link at a time). Indeed, one can ensure that each message reaches its destination in $O(\|\mathcal{D}\|)$ time as follows. A node z receiving a message with destination at distance d from z forwards it only after all messages of the protocol with a destination of distance more than d from z have been received and forwarded. By induction over the distance of a message from its destination, it is easily proven that this "Farthest-to-Go" queuing policy allows each message to traverse one link towards its destination after at most one time unit of waiting. Thus, the $O(\|\mathcal{D}\|)$ time complexity also holds for the entire algorithm. □

Instead of sending many small messages individually, we can choose to sequence the process of message sending so that messages for different failing edges are bundled before sending (see also [3,4] for applications of this idea). This leads to an alternative with fewer but longer messages:

Corollary 1. *After preprocessing, the* distributed all best swaps *problem can be solved using $O(n)$ messages of size $O(n)$ each, and $O(\|\mathcal{D}\|)$ time.*

5 The Preprocessing Phase

The preprocessing phase serves the purpose of making the needed terms in the sums described in the previous section available at the nodes of the tree.

In the preprocessing phase, a diameter $\mathcal{D}$ of T is chosen, and its two ends d_1 and d_k as well as its center d_c are identified. This can be done essentially by a convergecast, followed by a broadcast to distribute the result (see e.g. [9]); we omit the details. Hence, after preprocessing exchanges $O(n)$ messages, each node knows the information that is requested in (A) and (C) below. It is crucial that during preprocessing, each node obtains enough information to later carry out all computational steps to determine path components (i), (ii) and (iii). More precisely, each node gets the following global information (the same for all nodes):

(A) The endpoints d_1 and d_k of the diameter, the length $|\mathcal{D}|$ of the diameter, and the lengths $|\mathcal{D}_L|$ and $|\mathcal{D}_R|$.

(B) The length μ_R of a longest path starting in d_k that is fully inside $T_{d_{c+1}}$, together with the node ρ_R on $\mathcal{D}$ where such a path leaves the diameter. Figure 5 illustrates such a longest path μ_R. Moreover, the distance $d(\rho_R, d_c)$ must be known. Symmetrically, the length μ_L of a longest path starting in d_1 that is fully inside $T_{d_{c-1}}$, with the corresponding node ρ_L and distance $d(\rho_L, d_c)$ are required.

In addition, each node z obtains the following information that is specific for z:

(C) For each child $q \in C(z)$ of its children, the height T_q of q's subtree.
(D) Is z on the diameter $\mathcal{D}$, yes or no.
(E) The distance $d(z, d_c)$ of z to d_c.
(F) The identification of the parent $p(z)$ of z in T.
(G) To which of V_L, V_C and V_R does z belong.
(H) If $z \notin \mathcal{D}$, the closest predecessor u of z on the diameter; the distance $d(u, d_c)$ from u to d_c.
(I) If z is on the left (right) diameter $\mathcal{D}_L$ ($\mathcal{D}_R$), with $z = d_i$, the length $\lambda(d_i)$ of a longest path in T starting at d_c and neither containing the node d_{c+1} (d_{c-1}) nor the node d_{i-1} (d_{i+1}) (see Fig. 5).
(J) For each of the neighbours z' of z in G, which of V_L, V_C and V_R contains z'; the distance $d(z', d_c)$ from z' to d_c; the nearest predecessor u' of z' on $\mathcal{D}$, the distance $d(u', d_c)$.

Computing the Additional Information. Recall that the first preprocessing part ends with a broadcast that informs all nodes about the information described in (A) and (C). The second part of the preprocessing phase follows now.

A node z receiving the message about $\mathcal{D}$ can infer from the previous convergecast whether it belongs to $\mathcal{D}$ itself by just checking whether the paths from z to d_1 and d_k go through the same neighbour of z.

Information (E) is obtained by having the center node send a "distance from d_c" $d(d_c, d_*)$ message to both neighbours d_{c+1} and d_{c-1} on $\mathcal{D}$, which is forwarded and updated on the diameter. This information is used by the diameter nodes for computing $\lambda(d_i)$, required in (I). The center initiates the inductive computation of $\lambda(d_i)$:

- $\lambda(d_c)$ is the depth of a deepest node in V_C.
- For each d_j, $1 \leq j < c$, $\lambda(d_j) = \max\{\lambda(d_{j+1}), d(d_c, d_j) + h_2(d_j)\}$, h_2 being the height of a highest subtree of d_j apart from the diameter subtree.
- For each d_j, $c < j \leq k$, $\lambda(d_j) = \max\{\lambda(d_{j-1}), d(d_c, d_j) + h_2(d_j)\}$.

In order to compute μ_L and μ_R as required in (B), we define $\mu(d_i)$ for each node d_i on $\mathcal{D}_L$ as the length of a longest path starting in d_1 that is fully inside T_{d_i}, together with the node $\rho(d_i)$ on $\mathcal{D}_L$ where such a path leaves the diameter. For d_i on $\mathcal{D}_R$, the definition is symmetric. We then have $\mu_L = \mu(d_{c-1})$ and $\mu_R = \mu(d_{c+1})$. The inductive computation of $\mu(d_i)$ is started by d_1 and d_k, and then propagated along the diameter:

- $\mu(d_1) = \mu(d_k) = 0$;
- for each d_j, $1 < j < c$, $\mu(d_j) = \max\{\mu(d_{j-1}), d(d_1, d_j) + h_2(d_j)\}$;
- for each d_j, $c < j < k$, $\mu(d_j) = \max\{\mu(d_{j+1}), d(d_k, d_j) + h_2(d_j)\}$.

Along with $\mu(d_j)$, $\rho(d_j)$ and $d(\rho(d_j), d_c)$ can be computed as well. The computation stops in d_c, which receives the messages $(\mu(d_{c-1}), \rho(d_{c-1}), d(\rho(d_{c-1}), d_c)) = (\mu_L, \rho_L, d(\rho_L, d_c))$ and $(\mu(d_{c+1}), \rho(d_{c+1}), d(\rho(d_{c+1}), d_c)) = (\mu_R, \rho_R, d(\rho_R, d_c))$. Altogether, this second preprocessing part operates along the diameter and takes $O(\|\mathcal{D}(T)\|) = O(n)$ messages.

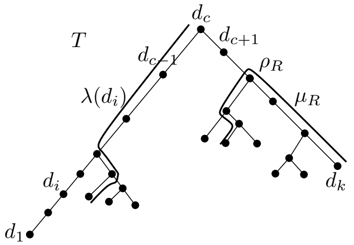

Fig. 5. Definition of $\lambda(d_i)$, μ_R and ρ_R

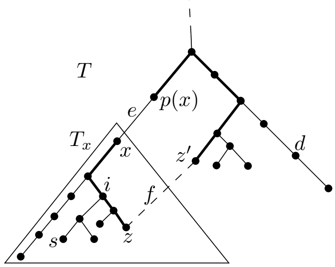

Fig. 6. Only some nodes need to know about failure of edge $e = (x, p(x))$

Distributing the Information. When the computation of $(\mu_L, \rho_L, d(\rho_L, d_c))$ and $(\mu_R, \rho_R, d(\rho_R, d_c))$ completes in d_c, the center packs these values plus the values $|\mathcal{D}_L|$ and $|\mathcal{D}_R|$ into one message M_*. It adds the appropriate one of the labels "V_L","V_R" and "V_C" to M_*, before forwarding M_* to d_{c-1}, d_{c+1} and any other neighbour of d_c in T and then flooding the tree. Additionally, M_* contains the "distance from d_c" information which is updated on forwarding, such that all nodes know their distance to the center[2]. When M_* is forwarded from a node $u \in \mathcal{D}$ to a node not on $\mathcal{D}$, it is extended by the "distance from u" information, which is also updated on forwarding. In addition, $d(u, d_c)$ is appended to M_*. Finally, if node z receives M_* from node v, then z learns that v is its parent.

At the end of this second part of the preprocessing phase, each node z' sends a message M' to each of its neighbours z in $G \backslash T$. Note that this is the only point in our solution where messages need to be sent over edges in $G \backslash T$. M' contains $d(z', d_c)$ and exactly one of { "$z' \in V_L$", "$z' \in V_C$" , "$z' \in V_R$" }, whichever applies. Furthermore, let u' be the nearest ancestor of z' on $\mathcal{D}$; the distance $d(u', d_c)$ is also appended to M'.

As a consequence, after each node has received its version of the message M_*, the information stated in (B), (E), (F), (G), (H) is known to each node. Furthermore, each node that has received M' from all its neighbours in G knows the information stated in (J). The distribution of this information requires $O(\|\mathcal{D}(T)\|)$ time and $O(m)$ messages. Let us summarize.

Lemma 2. *After the end of the two parts of the preprocessing phase, which requires $O(\|\mathcal{D}\|)$ time, all nodes know all information $(A) - (J)$, and $O(m)$ messages have been exchanged.*

Recognizing Swap Edges Using Labels. A node $v \in T_x$ must be able to tell whether an incident edge $f = (v, w)$ is a swap edge for $e = (x, p(x))$ or not. We achieve this by the folklore method of numbering nodes in two ways, a preorder traversal and a reverse preorder traversal. After this, a node can decide in constant time whether an edge is a swap edge. For details, see [3,4].

[2] The nodes on $\mathcal{D}$ already have that information at this point, but all other nodes still require it.

6 Routing Issues

A natural question arises concerning routing in the presence of a failure: After replacing the failing edge e by a best swap edge f, how do we adjust our routing mechanism in order to guide messages to their destination in the new tree $T_{e/f}$? And how is routing changed back again after the failing edge has been repaired? Clearly, it is desirable that the adaptation of the routing mechanism is as fast and inexpensive as possible.

Existing Approaches. The simplest routing scheme uses a routing table of n entries at each node, which contains, for each possible destination node, the link that should be chosen for forwarding. This approach can be modified to allow swaps by storing additional n entries for the swap links at each node [3]. In [5] a scheme is proposed that stores only one swap entry, at the cost of choosing suboptimal swap edges. All these approaches require $O(n^2)$ routing entries in total.

In the following, we propose to use a *compact* routing scheme for arbitrary trees (shortest paths, minimum diameter, or any other) which requires only δ entries, i.e. $\delta c \log n$ bits, at a node of degree δ, thus n entries or $mc \log n$ bits in total, which is the same amount of space that the *interval routing* scheme of [10] requires. The header of a message requires $c \log n$ bits to describe its destination.

Our Routing Scheme. Our routing scheme for trees is based on the labelling $\gamma : V \to \{1, \ldots, n\}^2$ described in the end of Section 5. Note that γ allows to decide in constant time whether a is in the subtree of b (i.e., $a \in T_b$) for any two given nodes a and b.

> **Basic Routing Algorithm:**
> A node s routes message M with destination d as follows: (i) If $d = s$, M has arrived at its destination. (ii) If $d \notin T_s$, s sends M to $p(s)$. (iii) Otherwise, s sends M to the child $q \in C(s)$ for which $d \in T_q$.

This algorithm clearly routes each message directly on its (unique) path in T from s to d. Before describing the adaptation in the presence of a swap, observe that a node s which receives a message M with destination d can locally decide whether M traverses a given edge $e = (x, p(x))$: edge e is used by M if and only if exactly one of s and d is in the subtree T_x of x, i.e., if $(s \in T_x) \neq (d \in T_x)$. Thus, it is enough to adapt routing if all nodes are informed about a failing edge (and later the repair) by two broadcasts starting at its two incident nodes (the points of failure). However, the following lemma shows that optimal rerouting is guaranteed even if only those nodes which lie on the two paths between the points of failure and the swap edge's endpoints are informed about failures, which allows "piggybacking" all information for routing adjustment on the first message arriving at the point of failure after the failure occurred.

Lemma 3. *Let $e = (x, p(x))$ be the failing edge, and $f = (z, z')$ the best swap of e, where z is in T_x and z' in $T \backslash T_x$, as shown in Figure 6. If all nodes on the path from x to z know that e is unavailable and that $f = (z, z')$ is a best swap* *edge, then any message originating in $s \in T_x$ will be routed on the direct path from s to its destination d. Symmetrically, if all nodes on the path from $p(x)$ to z' know about e and f, then any message originating in $s \in T \backslash T_x$ will be routed on the direct path from s to its destination d.*

Proof. Let M be any message with source $s \in T_x$. If $d \in T_x$, then trivially M will be routed on its direct path, because it does not require edge e. If $d \in T \backslash T_x$, consider the path $\mathcal{P}_T$ from s to d in T, and the path $\mathcal{P}_{T_{e/f}}$ from s to d in $T_{e/f}$. Consider the last common node i of $\mathcal{P}_T$ and $\mathcal{P}_{T_{e/f}}$ in T_x. The path composed of the paths $\langle x, \ldots, i \rangle$, $\langle i, \ldots, z \rangle$ is exactly the unique path in T from x to z, so node i lies on that path.

Obviously, M will be routed on the direct path towards d up to i. As i lies on the path from x to z, it knows about the failure and the swap, and will route M towards z. The lemma assumes that any node on the path from i to z also knows about the swap. Thus, such nodes will route M on the direct path to z. At z, M will be routed over the swap edge f, and from z' on M is forwarded on the direct path from z' to d. □

Given Lemma 3, we propose the following "lazy update" procedure for informing nodes about an edge failure:

> **Algorithm** Swap**:**
> If an edge fails, no action is taken as long as no message needs to cross it. As soon as a message M which should be routed over the failing edge arrives at the point of failure, information about the failure and its best swap is attached to message M, and M is routed towards the swap edge. On its way, all nodes which receive M route it further towards the swap edge, and remember for themselves the information about the swap.

Observation (Adaptivity). *After one message M has been rerouted from the point of failure to the swap edge, all messages originating in the same side of T as M (with respect to the failing edge) will be routed to their destination on the direct path in the tree (i.e., without any detour via the point of failure).*

If a failing edge has been replaced by a swap edge, then all nodes which know about that swap must be informed when the failure has been repaired. Therefore, a message is sent from the point of failure to the swap edge (on both sides if necessary), to inform these nodes, and to deactivate the swap edge.

7 Discussion

We have presented a distributed algorithm for computing all best swap edges for a minimum diameter spanning tree. Our solution is asynchronous, requires unique identifiers from a linearly ordered universe (but only for tiebreaking to determine a center node), and uses $O(\|\mathcal{D}\|)$ time and $O(\max\{n^*, m\})$ small messages, or $O(n)$ messages of size $O(n)$. It remains an open problem to extend our approach to subgraphs with other objectives; for instance, can we efficiently compute swap edges for failing edges in a spanner?

References

1. Andrews, M., Awerbuch, B., Fernández, A., Leighton, T., Liu, Z., Kleinberg, J.: Universal-stability results and performance bounds for greedy contention-resolution protocols. J. ACM 48(1), 39–69 (2001)
2. Bui, M., Butelle, F., Lavault, C.: A Distributed Algorithm for Constructing a Minimum Diameter Spanning Tree. Journal of Parallel and Distributed Computing 64, 571–577 (2004)
3. Flocchini, P., Enriques, A.M., Pagli, L., Prencipe, G., Santoro, N.: Point-of-failure Shortest-path Rerouting: Computing the Optimal Swap Edges Distributively. IEICE Transactions on Information and Systems E89-D(2), 700–708 (2006)
4. Flocchini, P., Pagli, L., Prencipe, G., Santoro, N., Widmayer, P.: Computing All the Best Swap Edges Distributively. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544. pp. 154–168. Springer, Heidelberg (2005)
5. Ito, H., Iwama, K., Okabe, Y., Yoshihiro, T.: Single Backup Table Schemes for Shortest-path Routing. Theoretical Computer Science 333(3), 347–353 (2005)
6. Nardelli, E., Proietti, G., Widmayer, P.: Finding All the Best Swaps of a Minimum Diameter Spanning Tree Under Transient Edge Failures. Journal of Graph Algorithms and Applications 5(5), 39–57 (2001)
7. Nardelli, E., Proietti, G., Widmayer, P.: Swapping a Failing Edge of a Single Source Shortest Paths Tree Is Good and Fast. Algorithmica 35(1), 56–74 (2003)
8. Di Salvo, A., Proietti, G.: Swapping a Failing Edge of a Shortest Paths Tree by Minimizing the Average Stretch Factor. In: Kralovic, R., Sýkora, O. (eds.) SIROCCO 2004. LNCS, vol. 3104. Springer, Heidelberg (2004)
9. Santoro, N.: Design and Analysis of Distributed Algorithms. Wiley Series on Parallel and Distributed Computing. Wiley, Chichester (2007)
10. Santoro, N., Khatib, R.: Labelling and Implicit Routing in Networks. The Computer Journal 28(1), 5–8 (1985)