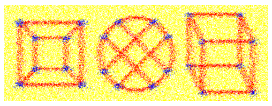
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 14, no. 2, pp. 287–306 (2010)

Computing All Best Swaps for Minimum-Stretch Tree Spanners

Shantanu Das[1] *Beat Gfeller*[2] *Peter Widmayer*[3]

[1]LIF, Aix-Marseille University, France.
[2]IBM Research, Zurich, Switzerland
[3]Institute of Theoretical Computer Science, ETH Zurich, Switzerland

Abstract

In a densely connected communication network, represented by a graph G with non-negative edge weights, it is often advantageous to route all communication on a sparse spanning subnetwork, typically a spanning tree of G. We consider a tree spanner T of G which guarantees that for any two nodes, their distance in T is at most k times their distance in G, where k, called the stretch, is as small as possible. When an edge of the communication tree T fails, network functionality may be restored by re-connecting the two separated parts of the tree with a swap edge. In situations where the failure can be repaired rapidly, such a quick fix is preferred over the recomputation of an entirely new minimum-stretch tree, because it is much closer to the previous solution and hence requires far fewer adjustments in the routing scheme. We are therefore interested in the problem of finding for any possibly failing edge in the spanner T, a best swap edge that minimizes the stretch of the new tree. We show how all these best swap edges can be computed in total time $O(m^2 \log n)$ in graphs with arbitrary non-negative edge weights. For graphs with unit weight edges, we present an $O(n^3)$ time algorithm. Furthermore, we present a distributed algorithm for computing the best swap for each edge in the tree spanner.

Submitted:	Reviewed:	Accepted:	Final:	Published:
September 2009	April 2010	April 2010	April 2010	June 2010
	Article type: Regular paper		Communicated by: U. Brandes	

A preliminary version of this paper was presented at the 19th International Symposium on Algorithms and Computation (ISAAC 2008) [3].

This work was partially supported by the National Competence Center in Research on Mobile Information and Communication Systems NCCR-MICS, a center supported by the Swiss NSF under grant number 5005 – 67322, and by the Swiss SBF under contract no. C05.0047 within COST-295 (DYNAMO) of the European Union.

E-mail addresses: shantanu.das@acm.org (Shantanu Das) bgf@zurich.ibm.com (Beat Gfeller) widmayer@inf.ethz.ch (Peter Widmayer)

1 Introduction

1.1 Minimum Stretch Tree Spanners

In a typical communication network, there are often more links (i.e., communication channels) available than what is useful for providing most services. The presence of these additional links makes it possible to deal with failures in the network. For routing efficiency, it is beneficial to maintain a connected sub-network (i.e. a subset of all available links) and route all communication through this subnet. We represent the original network as a connected, undirected graph $G = (V, E)$, and the subnet as a spanning tree T of the graph G. Instead of using any arbitrary spanning tree of G, we prefer one which has certain desirable properties that is tailored towards the computations performed. In this paper, we measure the overhead for communication as a result of using the subnet T instead of the original network G as the largest multiplicative increase in distance that any pair of nodes experiences. Thus, we use a tree T which minimizes the maximum stretch between any two nodes in T, where the stretch between nodes $a, b \in T$ is the ratio of their distance in T over their distance in G. Such a tree is called an optimal tree spanner of G. Tree spanners are used for routing in communication networks because they achieve a good tradeoff between the lengths of communication paths and the sizes of routing tables needed [16].

A critical problem with which we are confronted in this context is what happens when one of the links, say edge $e \in T$, fails and thereby disconnects the tree. There are at least two possible (and extreme) solutions to this: (1) recomputing an entirely new optimal tree spanner for $G - e$, or (2) replacing just the failing edge e by another edge (called a swap edge) that connects the two disconnected parts of $T - e$ in a best possible way, i.e., so that the stretch of the resulting tree is as small as possible. For temporary network failures, the second approach is much better suited than the first, because it is more efficient to use a swap edge for the duration of the failure, so that we can quickly revert back to the original spanner T once the fault has been repaired. Furthermore, this approach needs only a very small adjustment of routing tables and has therefore attracted research attention in recent years under the name of "on-the-fly rerouting" [7, 9, 11]. As an aside, note also that an entirely new optimal tree spanner might not only require a total replacement of all routing table entries, but is in addition NP-hard to find.

When choosing a best possible swap edge for a failing edge e, it is natural to use the same criterion as before, i.e. to minimize the stretch of the resulting tree. We always measure the stretch of a tree with respect to distances in the original graph G, and not with respect to distances in the fault-free subgraph $G - e$. This is due to two different reasons: First, this definition is more stable in the sense that the stretch of a tree does not depend on the failing edge that it replaces. Second, measuring the stretch with respect to distances in $G - e$ would have the unnatural effect that for some failing edges, the stretch of the swap tree might actually be lower than the stretch of the optimal tree spanner of G.

Interestingly enough, by merely going for the best swap, for unweighted graphs we are guaranteed to find a tree that is not all that bad even in comparison with an entirely new optimal tree spanner: We show that the stretch of a new tree T' obtained by adding a best swap edge is at most twice that of an optimal tree spanner of $G - e$ (again, measured w.r.t. distances in G). In order to quickly recover from an arbitrary edge failure, we precompute the best swap edge for each possible failing edge. The problem we consider in this paper is that of efficiently computing for every edge in the tree T a best swap edge. This *All-Best-Swaps* (ABS) problem has been studied for the cases when the tree T is a minimum spanning tree (MST), a shortest paths tree (SPT), or a minimum diameter spanning tree (MDST), with the corresponding different optimization criteria.

Our paper is the first to study the problem of finding all best swaps for an optimal tree spanner. This problem appears to be considerably more difficult than the previously studied ones: In all of the latter, one can evaluate a tree obtained by replacing a failing edge with a swap edge candidate in constant time, after some suitable preprocessing. However, for evaluating the stretch of such a tree, one needs to consider (at least implicitly) the stretch of each pair of nodes in the graph, which seems impossible to do in constant time, unless an expensive preprocessing step is used. Furthermore, in the previously studied problems, the quality of a given swap edge could be described somewhat independently of the particular failing edge they are going to replace. Again, this is no longer possible when evaluating swap edges for tree spanners. For the above reasons, none of the techniques used in earlier studies are directly applicable to our problem.

1.2 Our Contributions.

We first present and analyze a brute-force algorithm for solving the problem in Section 2.2. This algorithm requires $O(m^2 n)$ time for a graph having n nodes and m edges. In Section 3, we describe a more efficient algorithm that reduces the time complexity to $O(m^2 \log n)$ and requires $O(m)$ space. We also present an $O(n^3)$ time and $O(n^2)$ space solution for unweighted graphs in Section 4. In Section 5, we show how to compute all the best swaps of a tree spanner in a distributed fashion, where initially no node has complete knowledge about the network topology. Our distributed algorithm solves the problem with a communication cost of $O(m \cdot D + n \log n)$ messages of size $O(\log^2 n)$ bits, where D is the diameter of T. Finally, in Section 6 we compare our approach of swapping faulty edges with the alternative of recomputing a new tree spanner for the graph excluding the faulty edge.

1.3 Related Work.

The concept of graph spanners was introduced by Peleg and Ullman [15] who used it to construct good synchronizers for communication networks. Later, Peleg and Upfal [16] showed that spanners are useful as subnets for routing as they optimize both the route lengths and the space required for storing routing

information. Graph spanners (and in particular sparse spanners) are useful in many applications such as designing communication networks, distributed systems, parallel computers and also in motion planning [2].

The problem of finding a tree spanner that minimizes the maximum stretch, called the MMST problem, was shown to be NP-hard [2]. It can be approximated withing a factor of $O(\log n)$ in unweighted graphs [6].

As mentioned before, the ABS problem has been studied earlier for different optimization criteria, where the original tree is either a minimum spanning tree (MST) or a shortest paths tree (SPT) or a minimum diameter spanning tree (MDST). For MSTs, finding all best swap edges is closely related to sensitivity analysis: The weight of the best swap for a tree edge e is equal to the threshold up to which the weight of e can be increased without forcing it out of the MST. Using this connection, in a model where edge weights are atomic, but side-computations on a RAM are allowed, all the best swap edges can be found in $O(m)$ expected time using a randomized algorithm given by Dixon *et al.* [5]. Furthermore, they present a deterministic algorithm with asymptotically optimal running time, but whose exact running time cannot be easily determined. The best known explicit bound for deterministic algorithms is $O(m \log \alpha(m, n))$, where, $\alpha(m, n)$ denotes a functional inverse of the Ackermann function [17].

When the original tree is a SPT and the objective is minimizing the (maximum or total) distance from a fixed root, there are several possible definitions for the best swap edge. For many of these definitions, the ABS problem could be solved in $O(m\alpha(m, n))$ time [13]. For another variant, an $O(m\alpha(m, m) \log^2 n)$ time algorithm was presented in a follow-up paper [1] of [4].

While the above solutions are centralized, there also exist distributed solutions for computing best swaps in both SPTs [9] and MSTs [8], where the efficiency is measured in terms of the communication cost. For MSTs, a stronger version of the problem was studied, where the failures were assumed to occur at the nodes of the network, thereby disabling several edges at the same time [8].

In the case of MDSTs, there exists a centralized solution [10] that requires $O(m \log n)$ time and $O(m)$ space. The distributed version of the problem has been solved [11] using $O(n^* + m)$ messages of constant size, where n^* is the size of the transitive closure of the tree, when the edges are directed towards the node initiating the computation.

2 Computing All Best Swaps

2.1 Some Definitions and Properties

We use the following definitions and notations throughout this paper. The communication network is considered to be a 2-edge-connected, undirected graph $G = (V, E)$, with $n = |V|$ nodes and $m = |E|$ edges. Each edge $e \in E$ has a non-negative real *weight* denoted by $|e|$. We assume that the weight of an edge is in the range $[0, M]$ where M is at most polynomial in n, i.e., we do not allow

arbitrarily large edge-weights[1].

For any path $\mathcal{P} = \langle e_1, \ldots, e_r \rangle$ in the graph G, the length $|\mathcal{P}|$ of the path is the sum of the weights of the edges $e_1, \ldots, e_r$. The *distance* $d_H(x, y)$ between any two nodes x, y in a subgraph H of G is the length of a shortest path in H between x and y.

Definition 1 *For any two nodes $x, y \in V(G)$, the* stretch *of (x,y) in a spanning tree T of G is the ratio given by*

$$Stretch_T(x, y) := d_T(x, y)/d_G(x, y).$$

The stretch of the spanning tree T is the maximum stretch for any pair of nodes in T, i.e.

$$Stretch(T) := \max_{x,y \in V} \{d_T(x, y)/d_G(x, y)\}.$$

An optimal tree spanner *is a spanning tree with minimum stretch.*

If the given spanning tree $T = (V, E_T)$ is rooted at some fixed node u, then for each node $x \neq u$, we denote the *parent* of x by $p(x)$ and the set of its *children* by $C(x)$. Furthermore, T_x denotes the subtree of T rooted at x, including x.

The removal of any edge $e = (x, y)$ from T partitions the spanning tree into two disjoint trees $T^{x\setminus y}$ (the subtree containing x but not y) and $T^{y\setminus x}$ (the subtree containing y but not x). A *swap edge* f for e is any edge in $E\setminus E_T$ that (re-)connects $T^{x\setminus y}$ and $T^{y\setminus x}$, i.e., for which $T_{e/f} := (V, E_T\setminus\{e\} \cup \{f\})$ is a spanning tree of $G - e := (V, E\setminus\{e\})$. 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 the stretch of $T_{e/f}$, defined as $\max_{x,y \in V}\{d_{T_{e/f}}(x, y)/d_G(x, y)\}$, is minimum. Any edge $f \in E\setminus E_T$ is called a *candidate swap edge* (or *non-tree edge*), as it is a swap edge for at least one edge in T.

Definition 2 *The* All-Best-Swaps *problem for a given graph G and a given optimal tree spanner T of G consists of finding for every edge $e \in E_T$ a best swap edge.*

When we replace an edge $e = (x, y)$ in the optimal tree spanner T by a swap edge f, the stretch of a pair of nodes (a, b) remains the same in the new tree $T_{e/f}$ if either both a and $b \in T^{x\setminus y}$ or, both $a, b \in T^{y\setminus x}$. The following property of tree spanners, follows from a known result (Lemma 5.1 in [6]).

Property 1 *Let $G = (V, E)$ be any weighted, undirected graph and let T be a spanning tree G. For any pair of nodes $a, b \in V$, there exists an edge $f = (x, y) \in E$ such that $|f| = d_G(x, y)$ and $Stretch_T(x, y) \geq Stretch_T(a, b)$.*

[1]The size of the edge weights is important only for the distributed version of the problem (see Section 5) where the communication between processors must be kept small.

Proof: If (a, b) is an edge of G whose weight is at most the length of any other path from a to b, then there is nothing to prove. Otherwise, consider some shortest path between a and b in G that consists of the edges $e_1, e_2, \ldots e_k$ where $e_i = (x_i, y_i)$, $x_1 = a$, $x_{i+1} = y_i$, $\forall i, 1 \leq i < k$ and $y_k = b$. Note that each edge e_i in the path satisfies the condition that $|e_i| = d_G(x_i, y_i)$ (otherwise we can replace that edge to obtain a shorter path). We have

$$\frac{d_T(a,b)}{d_G(a,b)} \leq \frac{\sum_i d_T(x_i,y_i)}{\sum_i |e_i|} = \frac{\sum_i Stretch_T(x_i,y_i) \cdot |e_i|}{\sum_i |e_i|}$$

Thus, the stretch of (a, b) is not larger than the weighted average of the stretches of the pairs (x_i, y_i). Hence it must be true that for at least one such i, $Stretch_T(x_i, y_i) \geq Stretch(a, b)$. This (x_i, y_i) pair corresponds to the edge $f \in E$ in the lemma. □

From the above discussion, it follows that for computing the stretch of a swap tree, it is sufficient to consider the stretch of only those pairs of nodes (x, y) where $(x, y) \in E$. Accordingly, we define the concept of relevant stretch pairs, which will be used in the description of our algorithms:

Definition 3 *Each pair of nodes a, b with $(a, b) \in E$ is called a* stretch pair*. A stretch pair $g = (a, b)$ is said to be* relevant *for a given failing edge e if the path from a to b in T contains the edge e (in other words, if g is also a swap edge for e).*

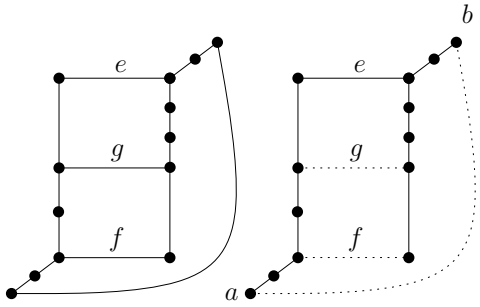

Figure 1: An example showing that minimizing detour length does not minimize the stretch: On the left side, the 2-edge-connected graph G is shown, and on the right side the given tree spanner with stretch 8 is shown. Assuming that all edges have equal weight, the swap edge f minimizes the stretch to the value 9. However, the swap edge minimizing the detour length is g, which yields a stretch of 10 (attained by the stretch pair (a, b)), worse than choosing swap edge f.

2.2 Naive Approach

To compute the best swap edge for an edge $e \in T$, we need to compare the $\Omega(m)$ possible candidate swap edges that are relevant for the failing edge e.

Unfortunately, there is no straightforward way of selecting the best among these candidates without evaluating each possible candidate. A simple trick, such as choosing the swap edge minimizing the detour around the failure, typicall does not yield an optimal solution. For instance, see the counterexample shown in Figure 1. This example can be generalized to obtain an arbitrarily large difference between the stretch for the best swap f and the minimum detour edge g.

We first consider the brute-force method for solving the *All-Best-Swaps* problem in a tree spanner. For each edge e of T, we can simply consider each relevant swap edge f, compute the stretch of $T_{e/f}$ and select a swap edge with smallest stretch as the best swap edge for e. Notice that there could be $\Omega(m)$ relevant swap edges for each edge $e \in T$. Thus, the algorithm would iterate over $\Omega(nm)$ pairs (e, f) of failing edges and corresponding relevant swap edges. The running time of this approach clearly depends on how fast the stretch of a given tree $T_{e/f}$ can be computed. Due to Property 1 we know that for computing the stretch of $T_{e/f}$, it is sufficient to consider only those stretch pairs (a, b) where a and b are adjacent in G, as opposed to all $O(n^2)$ pairs of nodes in V. Thus the stretch of the tree $T_{e/f}$ can be computed in $O(m)$ time, if the stretch of each $(a, b) \in E$ can be computed in constant time. This can be done using some preprocessing as explained below.

First, we obtain distances between all pairs of nodes in G in time $O(nm + n^2 \log n)$, using the standard "all-pairs shortest paths" algorithm. Next, we root the tree T at an arbitrary node r and compute the "to-root" distance $d_T(r, v)$ for each node $v \in V(G)$, with a single preorder traversal of T. Finally, we construct a data structure which provides the nearest common ancestor (nca) of any two given nodes in constant time. Such a data structure can be computed in $O(n)$ time, for example using the method described in [12].

After the preprocessing, we consider each relevant stretch pair (u, v), i.e. each $(u, v) \in E \backslash E_T$ where u and v lie on different sides of the failing edge[2]. For each such pair (u, v) we compute the distance in $T_{e/f}$ as

$$d_{T_{e/f}}(u, v) = d_T(u, p) + |f| + d_T(q, v),$$

where $f = (p, q)$ and p lies on u's side of the cut induced by e. Further, $d_T(u, p)$ (and similarly $d_T(v, q)$) is computed as

$$d_T(u, p) = d_T(u, \mathrm{nca}(u, p)) + d_T(p, \mathrm{nca}(u, p)).$$

Lemma 1 *After preprocessing in time $O(mn + n^2 \log n)$, for any failing edge $e \in E_T$, swap edge $f \in E \backslash E_T$, and relevant stretch pair (u, v), the stretch of (u, v) in $T_{e/f}$ can be computed in $O(1)$ time.*

Proof: Each of the five terms in the equations above can be computed in constant time. Note that the distance in the tree T between any node u and one of its ancestors, can be obtained as the absolute difference between the

[2]This can be checked using a "preorder/inverted preorder" labeling. For details, see e.g. [9], Section 3.2.

Algorithm 1: BestSwaps

```
for e ∈ T do
    Current-Best(e) ← ∞
for f = (u, v) ∈ E\T do
    Assign indices to the nodes in G
    Initialize Data Structure H
    for e_i ∈ Path_T(u, v) do
        Add to H stretch pairs in Start_i
        Remove from H stretch pairs in End_i
        St(e_i, f) ← GetMax(H)
        if St(e_i, f) < Current-Best(e_i) then
            Update Current-Best(e_i)
```

"to-root" distance of these two nodes (which can be looked up in the data structure that was precomputed). Finally, to obtain the stretch of (u, v), we divide $d_{T_{e/f}}(u, v)$ by their distance in G, which has already been precomputed. □

To summarize, $d_{T_{e/f}}(u, v)$ and thus the stretch of (u, v) can be computed in constant time, for each of the $O(m)$ relevant stretch pairs, for a particular e and f. This implies the following result:

Theorem 1 *The* All-Best-Swaps *problem in a tree spanner can be solved in* $O(nm^2)$ *time.*

In the next few sections, we present some techniques to reduce this time complexity.

3 A Faster Algorithm

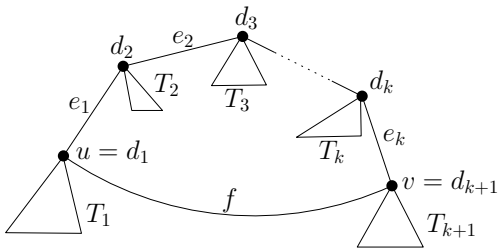

Figure 2: The cycle that a non-tree edge f forms with the given spanning tree.

In the following, we describe an algorithm which computes all best swap edges of a tree spanner in $O(m^2 \log n)$ time and $O(m)$ space. The idea of the

algorithm (called *BestSwaps*) is the following. We consider each potential swap edge $f \in E \backslash E_T$ separately, focusing on the cycle which $f = (u, v)$ forms with T (see Figure 2). This cycle consists of the edges $e_1, e_2, \ldots e_k$ which form a path in T. We use the nodes of V which lie on the path in T from $u = d_1$ to $v = d_{k+1}$ to partition T into subtrees as follows. With each node d_i on this path, we associate the subtree T_i, which consists of the connected component containing d_i in the graph $T \backslash \{e_1, e_2, \ldots, e_k\}$. For a given failing edge $e_i = (d_i, d_{i+1})$ for which f is a swap edge, the set of relevant stretch pairs contains all non-tree edges where one endpoint lies in some subtree $T_1, \ldots, T_i$, and the other endpoint lies in some subtree $T_{i+1}, \ldots, T_{k+1}$. We assign to each node the index i of the subtree T_i containing it. For any edge $(a, b) \in E \setminus E_T$ whose endpoints are a relevant stretch pair for f, this defines an order of the endpoints: if a's index is smaller than the index of b, we say that a is the *lower* endpoint, and that b is the *upper* endpoint. In order to evaluate f as a potential swap edge, we need to compute the stretch for every relevant stretch pair with respect to f and some failing edge e_i. Note that for $f = (u, v)$, the stretch of the pair a, b is given by $(d_T(a, u) + l(f) + d_T(b, v))/d_G(u, v)$, which is independent of the failing edge e_i.

We consider the potential failing edges $e_1, e_2, \ldots, e_k$, in that order and evaluate f as a potential best swap with respect to each e_i in turn. Observe the following: If $S(e_{i-1})$ is the set of relevant stretch pairs when considering f as a swap for e_{i-1}, then $S(e_i)$, the set of relevant stretch pairs when considering f as a swap for e_i, is $S(e_i) = \big(S(e_{i-1}) \cup \mathit{Start}_i\big) \setminus \mathit{End}_i$, where Start_i is the set of stretch pairs whose lower endpoint is d_i, and End_i is the set of stretch pairs whose upper endpoint is d_i. Therefore, we store the set $S(e_i)$ in a data structure H and update it as we move from e_i to e_{i+1}. To compute $S(e_i)$ from $S(e_{i-1})$, all stretch pairs that become relevant are added to H and all stretch pairs that become irrelevant are deleted from H. The data structure H we use to store the set $S(e_i)$ can be implemented as a heap where the priority of a stretch pair (a, b) is defined by the stretch value $\mathit{Stretch}_T(a, b)$. Notice that this stretch is independent of the failing edge e_i for a fixed swap edge f, and therefore the priority of a stretch pair stored in the heap need never be changed. The largest element in H yields the worst stretch pair for f replacing e_i. We simply check whether this value is smaller than the stretch of the current best swap edge for e_i (which we maintain in a separate data structure) and update the current swap edge for e_i if required. Once we have performed the above process for every edge $f \in E \setminus E_T$, we have obtained for each edge in T a best swap edge. Hence:

Theorem 2 *The algorithm* BestSwaps *computes all the best swap edges of a tree spanner in $O(m^2 \log n)$ time and using $O(m)$ space.*

Proof: We first show that the algorithm takes $O(m^2 \log n)$ time. For each swap edge $f = (u, v) \in E \setminus E_T$, the algorithm does the following. First, it assigns to each node z its index i, which denotes the subtree T_i in which it is contained, as described above. Assigning these indices takes $O(n)$ time. The non-tree edges of the graph G are partitioned into sets Start_i and End_i, corresponding to nodes d_i in the path from u to v. This can be done in $O(m)$ time. Further, for

each pair (f, e) of a swap edge f and a failing edge e, the algorithm performs a number of insertions and deletions[3] on the heap H. For a fixed swap edge f and all possible failing edges e_i, any stretch pair is inserted at most once and deleted at most once. Thus we require $O(m)$ heap operations, i.e. $O(m \log m)$ time for the process corresponding to each edge $f \in E \setminus E_T$. Note that we also need to compute the stretch of a stretch pair before inserting it, which however this takes only constant time using Lemma 1. Thus, in total the algorithm requires $O(m^2 \log m) = O(m^2 \log n)$ time.

As for the space requirements, the heap data structure H requires $O(m)$ space for storing at most m elements. Storing the current best for each edge $e \in T$ requires $O(n)$ space. □

4 An Algorithm for Unweighted Graphs

In this section we devote our attention to the special case of graphs with unit weight edges (henceforth called *unweighted* graphs). For such graphs, we present a dynamic programming algorithm for solving the ABS problem. The algorithm computes the best swap edge for each of the $n-1$ edges of T in a separate computation, each requiring $O(n^2)$ time and $O(n^2)$ space. For each failing edge $e = (l, r)$, we root the two trees $T^{l\setminus r}$ (for "left") and $T^{r\setminus l}$ (for "right") of $T-e$ at the nodes l and r, respectively. Recall that the stretch of a swap edge $f = (a, b)$ is obtained at some stretch pair x, y, whose stretch is $d_{T_{e/f}}(x, y)/d_G(x, y)$. In unweighted graphs, $d_G(x, y) = 1$ and hence the maximum stretch is obtained by the stretch pair x, y for which $d_{T_{e/f}}(x, y)$ is maximum. Furthermore, for $a, x \in T^{l\setminus r}$ and $b, y \in T^{r\setminus l}$ we have $d_{T_{e/f}}(x, y) = d_{T_{e/f}}(x, a) + |(a, b)| + d_{T_{e/f}}(b, y) = d_{T-e}(x, a) + |(x, y)| + d_{T-e}(b, y)$. Therefore, the stretch of a swap edge $f = (a, b)$ is equal to the length of a longest simple path from a to b in G, using only edges of $T-e$ plus either exactly one candidate swap edge $(x, y) \in E\setminus E_T$, or the edge e [4]. In the following, we call paths of this nature the *stretch paths* of the node pair a, b. In our approach, we compute the length of a longest stretch path for each of the $O(n^2)$ node pairs a, b, even for those which are not linked by an edge in G. It turns out that by partitioning the set of all stretch paths into nine different types, and by computing the length of the longest stretch paths of a particular type for each node pair a, b in a suitable order, all these lengths can be computed in $O(n^2)$ time by dynamic programming. In the following, we describe this approach in detail.

The *type* of a stretch path $\mathcal{P}$ depends on which of the edges incident to $a \in T^{l\setminus r}$ and $b \in T^{r\setminus l}$ it includes. If $\mathcal{P}$ contains the edge $(a, p(a))$, we say it goes *up* on the left side. If $\mathcal{P}$ contains an edge (a, q) for some $q \in C(a)$, we say it goes *down* on the left side. Furthermore, if $\mathcal{P}$ uses a candidate swap edge

[3] For the deletions, we assume that whenever an element is inserted into the heap, a pointer to its position in the heap is stored, such that the element can later be found in constant time and then removed in logarithmic time.

[4] We have to include e here because the stretch is measured with respect to G, not with respect to $G-e$.

incident to a (and hence does not contain any other edge from $T^{l\backslash r}$), we say it *stays* at a. The corresponding definitions hold for the right side of stretch paths. Hence, we have the following nine types of paths (where the first word corresponds to the left side of the path, and the second to the right side): `Stay-Stay`, `Stay-Down`, `Down-Stay`, `Stay-Up`, `Up-Stay`, `Down-Down`, `Down-Up`, `Up-Down`, `Up-Up`. For each `TypeA-TypeB` and each node pair a, b, we denote by `TypeA-TypeB`(a, b) the length of a longest stretch path from a to b of type `TypeA-TypeB`. If no stretch path from a to b of type `TypeA-TypeB` exists, then we define `TypeA-TypeB`$(a, b) := -\infty$.

We compute the longest path of each type with an inductive computation (dynamic programming) requiring $O(n^2)$ time. To that end, we first explain the necessary recursive equations. We start with `Stay-Stay` paths: for a given node pair a, b, the only possible path of that type is composed of the edge (a, b) (if present). Thus, we have

$$\texttt{Stay-Stay}(a,b) = \begin{cases} 1 & \text{if } (a,b) \in E\backslash E_T \cup \{e\} \\ -\infty & \text{otherwise.} \end{cases}$$

Clearly, `Stay-Stay`(a, b) for all $a, b \in V$ can be obtained in $O(n^2)$ time.

It is easy to see that the length of a longest stretch path of type `Stay-Down` satisfies the following recursion:

$$\texttt{Stay-Down}(a,b) = 1 + \max_{q \in C(b)} \Big\{ \max\{\texttt{Stay-Stay}(a,q), \texttt{Stay-Down}(a,q)\} \Big\}.$$

Naturally, the symmetric equation holds for `Down-Stay`(a, b). Note that this recursion can be translated into a dynamic program: as `Stay-Stay`(a, q) for any $a, q \in V$ is already available from the previous computation, we only need to ensure that `Stay-Down`(a, q) is available for all $q \in C(b)$ when `Stay-Down`(a, b) is computed. This is guaranteed if we consider the pairs a, b in an order in which the b's occur in postorder. Thus, `Stay-Down`(a, b) and `Down-Stay`(a, b), $\forall$ $a, b \in V$ can be computed in $O(n^2)$ time.

To compute all the `Stay-Up` paths, we need paths of type `Stay-Stay` as well as of type `Stay-Down`. More precisely:

$$\begin{aligned} \texttt{Stay-Up}(a,b) = \max \Big\{ \quad & 1 + \texttt{Stay-Stay}(a, p(b)), \quad 1 + \texttt{Stay-Up}(a, p(b)), \\ & 2 + \max_{q \in C(p(b)), q \neq b} \{\texttt{Stay-Down}(a,q), \texttt{Stay-Stay}(a,q)\} \quad \Big\}. \end{aligned}$$

A symmetric equation holds for `Up-Stay`(a, b). Assuming that `Stay-Stay` and `Stay-Down` have been previously computed for all pairs of nodes, we just need to guarantee that `Stay-Up`$(a, p(b))$ is available, when computing `Stay-Up`(a,b). So, in our dynamic programming algorithm, we consider the pairs a, b in an order where the b's occur in preorder. In this way, `Stay-Up`(a, b) and `Up-Stay`(a, b) $\forall$ $a, b \in V$ can be computed in $O(n^2)$ time.

Consider now a `Down-Down` stretch path from a to b (see Fig. 3(i)). We have:

$$\texttt{Down-Down}(a,b) = 1 + \max\Big\{ \max_{q\in C(a)} \{\texttt{Stay-Down}(q,b), \texttt{Down-Down}(q,b)\}, \max_{q'\in C(b)} \{\texttt{Down-Stay}(a,q'), \texttt{Down-Down}(a,q')\} \Big\}.$$

In order to write a dynamic program corresponding to this recursion, the node pairs a, b must be considered in an order such that all children of a node are considered before the node itself (i.e. both the trees T_l and T_r are traversed in postorder). In this way, $\texttt{Down-Down}(a,b)$ for all $a, b \in V$ can be computed in $O(n^2)$ time.

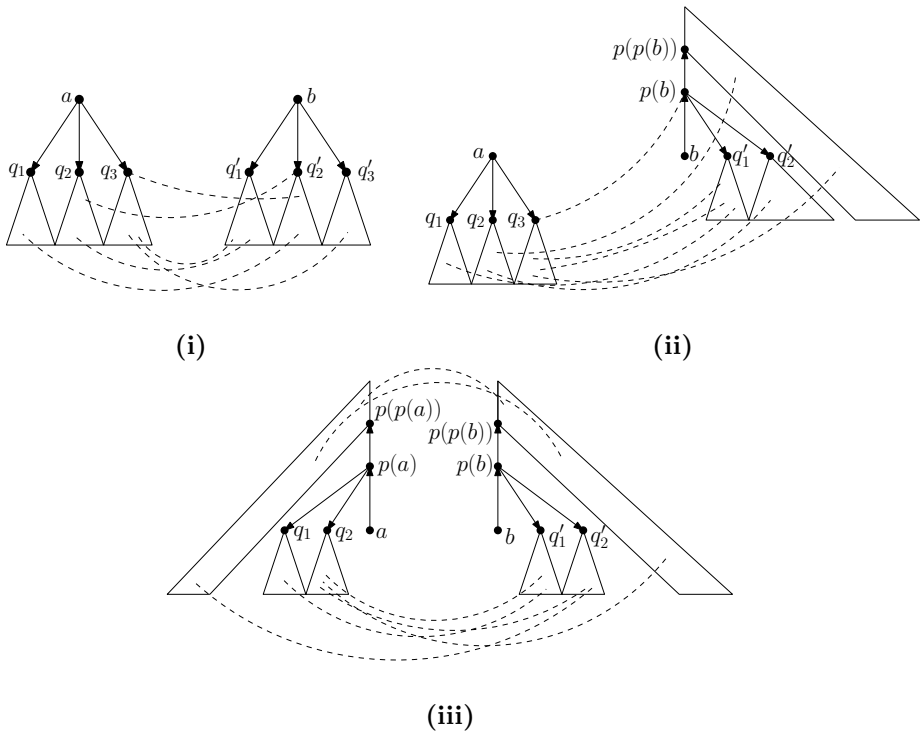

Figure 3: Three of the possible stretch path types of the node pair a, b: (i) `Down-Down`, (ii) `Down-Up`, (iii) `Up-Up`

Next, let us focus on the Down-Up paths (see Fig. 3(ii)). Here, we have[5]

$$\texttt{Down-Up}(a,b) = \max\Big\{ \quad 1+\texttt{Down-Stay}(a,p(b)), \quad 1+\texttt{Down-Up}(a,p(b)), \\ 2+\max_{q'\in C(p(b)),q'\neq b}\{\texttt{Down-Down}(a,q'),\texttt{Down-Stay}(a,q')\} \quad \Big\}.$$

We omit the equation for $\texttt{Up-Down}(a,b)$, which is completely symmetric. By considering all pairs $a,b\in V$ such that the b's occur in preorder, $\texttt{Down-Up}(a,b)$ and $\texttt{Up-Down}(a,b)$ can be computed in $O(n^2)$ time.

Finally, the length of a longest Up-Up stretch path for a,b can be expressed as (see Fig. 3(iii))

$$\begin{aligned}\texttt{Up-Up}(a,b) = \max\Big\{ \quad & 1+\texttt{Up-Stay}(a,p(b)), \quad 1+\texttt{Up-Up}(a,p(b)), \\ & 1+\texttt{Stay-Up}(p(a),b), \quad 1+\texttt{Up-Up}(p(a),b), \\ & 2+\max_{q'\in C(p(b)),q'\neq b}\{\texttt{Up-Down}(a,q'),\texttt{Up-Stay}(a,q')\}, \\ & 2+\max_{q\in C(p(a)),q\neq a}\{\texttt{Down-Up}(q,b),\texttt{Stay-Up}(q,b)\} \quad \Big\}.\end{aligned}$$

To obtain $\texttt{Up-Up}(a,b)$ for all $a,b\in V$ in $O(n^2)$ time, the pairs are considered in an order in which both the a's and the b's occur in preorder.

Each of these dynamic programs fills an $(n\times n)$-matrix, and thus needs $O(n^2)$ space. As mentioned in the beginning, we repeat these computations for each of the $n-1$ edges $e\in E_T$. Then, the algorithm computes, for each swap edge candidate $f=(u,v)$, the stretch of $T_{e/f}$ as

$$\max\Big\{ \quad \texttt{Stay-Stay}(u,v), \quad \texttt{Stay-Down}(u,v), \quad \texttt{Down-Stay}(u,v), \\ \texttt{Stay-Up}(u,v), \quad \texttt{Up-Stay}(u,v), \quad \texttt{Down-Down}(u,v), \\ \texttt{Down-Up}(u,v), \quad \texttt{Up-Down}(u,v), \quad \texttt{Up-Up}(u,v) \quad \Big\},$$

in constant time. After each computation, we can delete the computed matrix from memory, only storing the best swap edge found for the considered failing edge e. Thus, the total space complexity of our approach is $O(n^2)$. To summarize, we have the following result:

Theorem 3 *In unweighted graphs, all best swap edges of a tree spanner can be computed in $O(n^3)$ time and $O(n^2)$ space.*

5 A Distributed Solution

In this section, we consider the scenario where each node in the network initially has only local information about the network. We are interested in a distributed

[5] In the preliminary version [3], some terms in the expressions for $\texttt{Down-Up}(a,b)$ and $\texttt{Up-Up}(a,b)$ were inadvertently omitted.

algorithm which enables each node to compute the information it requires to modify its local routing table whenever any of the edges in T fails. More specifically, a node v having incident edges $e_1, e_2, \ldots, e_t \in E_T$ must compute the best swap edges $f_1, f_2, \ldots, f_t$ corresponding to these edges. Using this information, the routing mechanism described in [11] can adjust routing quickly when a failing edge is replaced by its best swap edge. Briefly, this works as follows: Each node of the tree is assigned a label as an identifier, and to each message, the label of its destination is attached. For each edge $e \in T$, only its two endpoints must know the best swap edge f for e (given by the labels of f's endpoints). When edge e fails, its two endpoints will notice the error and propagate an error message along the two paths leading from the two endpoints of e to the two endpoints of f. Any message which would normally use edge e will reach some node on this path, and from there will be rerouted directly through the swap f to its destination. More details of this rerouting mechanism are described in [11].

For the following, we consider the original tree spanner T to be rooted at some fixed node r, which is known to every node in the tree. Each node $x \in V$ also knows which of its incident edges connect to its children $C(x)$ in T and which edge connects to its parent $p(x)$ in T. We now describe our algorithm for the distributed computation of all best swaps.

During a preprocessing step, we assign labels to all nodes of the tree, which encode information that is later used for the distributed swap edge computation. In particular, given the labels of two nodes $a, b \in G$, it should be possible to compute the distance $d_T(a, b)$ as a function of the labels of a and b. There exists such a distance labeling for trees, in which the label $l(x)$ assigned to a node x consists of $O(\log^2 n)$ bits (see [14], Section 19.2, for example). Moreover, this labeling can be easily computed in a distributed fashion using $O(n \log n)$ messages of size $O(\log n)$ bits. At the end of the label computation, neighboring nodes in G can exchange label information using at most $2m$ messages. After the preprocessing step which requires $O(m + n \log n)$ messages, each node v knows the weight of each incident edge $f = (v, u)$ and the labels of the endpoints of f.

Each node $x \neq r$ in the tree is responsible for computing the best swap edge for the tree edge $e = (x, p(x))$. To that end, it must know the set of all potential swap edges for e, as well as the set of relevant stretch pairs for these swap edges. Note that these two sets are identical, and consist of exactly all those non-tree edges where one endpoint lies in T_x and the other endpoint lies in $T \backslash T_x$. We denote this set of edges by $Start(T, x)$. Each edge $f = (a, b) \in Start(T, x)$ is represented by the tuple $(l(a), l(b), |(a, b)|)$. We now show how the best swap edge for e can be computed using the information contained in $Start(T, x)$.

For any potential swap edge $f = (u, v)$ for e, and a relevant stretch pair (a, b), we have $\text{Stretch}_{T_{e/f}}(a, b) := d_{T_{e/f}}(a, b)/d_G(a, b)$. However, due to Property 1, for computing the stretch of $T_{e/f}$, it is sufficient to measure the stretch of a, b by $d_{T_{e/f}}(a, b)/|(a, b)|$. Note that this does not decrease the value of stretch for the worst stretch pair. Thus, if we take the maximum value of $\text{Stretch}(d_{T_{e/f}}(a, b))$ over all stretch pairs $(a, b) \in Start(T, x)$, we get the correct

value of Stretch($T_{e/f}$).

Assuming w.l.o.g. that $a, u \in T_x$, we have:

$$d_{T_{e/f}}(a,b) = d_T(u,a) + |(a,b)| + d_T(b,v)$$

As mentioned before, the value of $d_T(u,a)$ (respectively $d_T(b,v)$) can be computed from the labels $l(u)$ and $l(a)$ (resp. $l(b)$ and $l(v)$). Thus, given the list of edges $Start(T,x)$, each node x can locally compute the stretch of each swap candidate f for $e = (x, p(x))$, and thus obtains a best swap edge. The only part of the algorithm that remains to be described is the computation of $Start(T,x)$ at node x.

We can compute the set $Start(T,x)$ at each node x, using the convergecast technique on the tree, propagating information from the leaves of T up to the root, along the edges of T. For each leaf node $x_l \in T$, the set $Start(T,x_l)$ contains exactly all non-tree edges incident to x_l (and this information is already available at node x_l). An internal node $x_i \in T$ receives the set $Start(T,y)$ from each child y, and combines this information to compute the set $Start(T,x_i)$ as follows:

$$Start(T,x_i) = CS(x_i) \cup \{(x_i,v) \in E \setminus E_T : v \in V\}, \text{ where}$$

$$CS(x_i) = \bigcup_{y \in C(x_i)} (Start(T,y) \setminus \{(a,b) \in Start(T,y) : \text{nca}(a,b) = x_i\}).$$

If the two endpoints of an edge $f = (a,b)$ have node x as their nearest common ancestor, then such an edge appears in two of the sets received by node x from its children. Thus, node x can detect such edges and remove them from the list. Node x also adds all its incident non-tree edges to the list. Once the list $Start(T,x)$ has been computed at node x, this list is sent to $p(x)$.

During the algorithm, the only information exchanged between the nodes is the list of candidate swap edges. Further, the information about each swap edge $f = (u,v)$ travels only along the path in T from u to v. This accounts for at most $(m-n+1)D$ messages, where D is the diameter of the tree. Each message has a size proportional to the size of the node labels and edge weights. So the overall communication complexity of our algorithm is $O(m \cdot D + n \log n)$ messages, each of size $O(\log^2 n)$ bits[6].

6 Swapping versus Recomputing

In this section, we investigate how a best swap tree compares with a newly computed optimal tree spanner of $G - e$, with respect to the maximum stretch. We show that at least for unweighted graphs, the stretch is at most twice as large in the swap tree as in the tree spanner.

[6]Note that the preliminary version [3] claimed a different bound which, however, is not guaranteed by our algorithm.

Lemma 2 *For any failing edge e of an optimal tree spanner of an* unweighted *graph $G = (V, E)$, the maximum stretch of the* swap tree*, measured w.r.t. distances in G, is at most two times larger than the stretch of an optimal tree spanner of $G - e$. The bound of two is tight.*

Proof: Let T be the optimal tree spanner of G, let k be the maximum stretch of T, and let T' be the best swap tree when $e = (u, v)$ fails. Let (a, b) be a stretch pair for which the stretch with respect to T' is maximum, i.e.

$$(a, b) = \arg \max_{(i,j) \in E} \frac{d_{T'}(i, j)}{d_G(i, j)}.$$

Further, let (u', v') be the swap edge for e in T'. Since (a, b), $(u', v') \in E$, we know that $d_G(a, b) = d_G(u', v') = 1$. Notice that the path from a to b in T' can be obtained as a subset of the edges from the path between a and b in T, the path between u' and v' in T, and the new edge (u', v'). In other words, we have an upper bound on the distance $d_{T'}(a, b)$

$$d_{T'}(a, b) \leq d_T(a, b) - 2|(u, v)| + d_T(u', v') + |(u', v')|$$

This implies,

$$\begin{aligned} \frac{d_{T'}(a, b)}{d_G(a, b)} &\leq \frac{d_T(a, b) - 2|(u, v)| + d_T(u', v') + |(u', v')|}{d_G(a, b)} \\ &\leq \frac{d_T(a, b)}{d_G(a, b)} + \frac{d_T(u', v')}{d_G(a, b)} - 1 \\ &\leq k + \frac{d_T(u', v')}{d_G(u', v')} \quad \leq 2k. \end{aligned}$$

For any other spanning tree of G (including the optimal spanner of $G - e$), the stretch must be at least k, and hence the result follows. An example that achieves the bound of two is shown in Figure 4. □

7 Conclusions

The technique of "on-the-fly rerouting", where faulty edges in the routing tree are immediately replaced by the best available swap edge, has received a lot of attention in recent years. For the simple cases where the routing tree is either a minimum spanning tree (MST), a shortest path tree (SPT) or a minimum diameter spanning tree (MDST), the replacements for all possible failures can be computed together in a single efficient precomputation. However, the problem becomes more challenging if the original tree is a optimal tree spanner and the objective is to minimize the stretch factor in the resulting swap tree. For this case, we solved the *All-Best-Swaps* problem using $O(m^2 \log n)$ time and $O(m)$ space for weighted graphs and $O(n^3)$ time and $O(n^2)$ space for unweighted

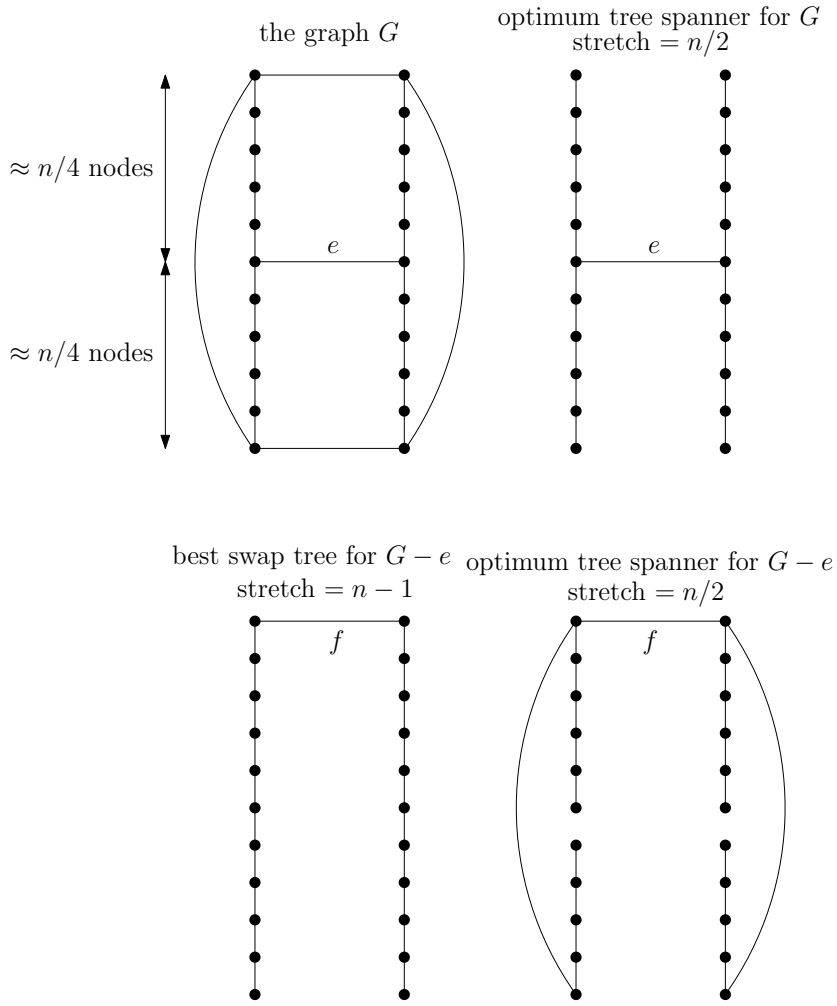

Figure 4: An example of a graph where the stretch of the best swap tree is two times worse than the best newly computed tree spanner.

graphs, where n and m are respectively the number of nodes and edges in the graph. We also showed how to solve the problem in a distributed fashion such that each node in the network obtains information about the best swap edges corresponding to each of its incident tree edges. In the event of an edge failure, this information can be used to quickly adjust the routing with local changes to the routing tables.

References

[1] D. Bilò, L. Gualà, and G. Proietti. Finding All the Best Swap Edges of a Routing Tree: a Faster Algorithm and an Effectiveness Analysis. Manuscript, 2008.

[2] L. Cai and D. G. Corneil. Tree Spanners. *SIAM J. Discr. Math.*, 8(3):359–387, 1995.

[3] S. Das, B. Gfeller, and P. Widmayer. Computing Best Swaps in Optimal Tree Spanners. In *19th International Symposium on Algorithms and Computation (ISAAC)*, volume 5369 of *LNCS*, pages 716–727. Springer, 2008.

[4] A. Di Salvo and G. Proietti. Swapping a failing edge of a shortest paths tree by minimizing the average stretch factor. *Theor. Comput. Sci.*, 383(1):23–33, 2007.

[5] B. Dixon, M. Rauch, and R. Tarjan. Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.

[6] Y. Emek and D. Peleg. Approximating Minimum Max-Stretch spanning Trees on unweighted graphs. In *SODA '04: Proceedings of the fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 261–270, 2004.

[7] P. Flocchini, A. M. Enriques, L. Pagli, G. Prencipe, and N. Santoro. Point-of-failure Shortest-path Rerouting: Computing the Optimal Swap Edges Distributively. *IEICE Transactions on Information and Systems*, E89-D(2):700–708, 2006.

[8] P. Flocchini, T. M. Enriquez, L. Pagli, G. Prencipe, and N. Santoro. Distributed Computation of All Node Replacements of a Minimum Spanning Tree. In *Proceedings 13th International Euro-Par Conference*, volume 4641 of *LNCS*, pages 598–607. Springer, 2007.

[9] P. Flocchini, L. Pagli, G. Prencipe, N. Santoro, and P. Widmayer. Computing all the best swap edges distributively. *J. Parallel Distrib. Comput.*, 68(7):976–983, 2008.

[10] B. Gfeller. Faster Swap Edge Computation in Minimum Diameter Spanning Trees. *Algorithmica*, 2010. To appear.

[11] B. Gfeller, N. Santoro, and P. Widmayer. A Distributed Algorithm for Finding All Best Swap Edges of a Minimum Diameter Spanning Tree. *IEEE Transactions on Dependable and Secure Computing*, 2009. To appear.

[12] D. Harel and R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[13] E. Nardelli, G. Proietti, and P. Widmayer. Swapping a Failing Edge of a Single Source Shortest Paths Tree Is Good and Fast. *Algorithmica*, 35(1):56–74, 2003.

[14] D. Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[15] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, pages 77–85, New York, NY, USA, 1987. ACM.

[16] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

[17] S. Pettie. Sensitivity Analysis of Minimum Spanning Trees in Sub-Inverse-Ackermann Time. In *Proceedings 16th Int'l Symposium on Algorithms and Computation (ISAAC)*, pages 964–973, 2005.