Computing best swaps in optimal tree spanners

Computing Best Swaps in Optimal Tree Spanners

ETH Technical Report 607

Shantanu Das, Beat Gfeller, Peter Widmayer

Institute of Theoretical Computer Science, ETH Zurich, Switzerland

Abstract. In a densely connected communication network, represented by a graph G with nonnegative edge-weights, it is often advantageous to route all communication on a sparse, spanning subnetwork, typically a spanning tree of G. With the communication overhead in mind, we consider a spanning tree 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. Such a spanning tree which minimizes the stretch is called an optimal tree spanner, and it can be used for efficient routing. However, for a communication tree, the failure of an edge is catastrophic; it disconnects the tree. Functionality can be restored by connecting both parts of the tree with another edge, while leaving the two parts themselves untouched. In situations where the failure can be repaired rapidly, such a quick fix is preferred over the recomputation of an entirely new optimal tree spanner, 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 to replace it. The objective here is naturally to minimize 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 nonnegative edge weights. For graphs with unit weight edges (also called unweighted graphs), we present an $O(n^3)$ time algorithm. Furthermore, we present a distributed algorithm for computing the best swap for each edge in the spanner.

1 Introduction

In a typical communication network, there are often more links (i.e., communication channels) available than what is useful for performing most computations. The presence of these additional links makes it possible to deal with failures in the network. However, at any given time, only a subset of the links are actually used for communication. For efficiency reasons, it is beneficial to maintain such a sub-network and route all communication through this subnet. We represent the original network as a connected, undirected graph $G = (V, E)$, and the subnet is a spanning tree T of this graph G. Instead of using any arbitrary spanning tree of G, we prefer one which has certain desirable properties supporting the required

computations. 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 [14].

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, thereby disconnecting 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 for simpler spanning trees, under the name of "on-the-fly rerouting" [5, 7, 9]. 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 swap edge for a failing edge e, it is natural to use the same criterion as before, that is, minimizing the stretch. 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 (transient) fault-free subgraph $G - e$ [1]. 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 also 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 pre-compute 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 one 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 particular, none of the techniques used in earlier studies are applicable to our problem.

[1] Notice that this is a more stable definition for the stretch of a tree.

Our Contributions. We first present and analyze a brute-force algorithm for solving the problem in Section 2. This algorithm requires $O(m^2n)$ time for a graph having n vertices 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. Finally, in Section 5, we show how to compute all the best swaps of a tree spanner in a distributed fashion. Our distributed algorithm solves the problem in $O(D)$ time with a communication cost of $O(n^* \log n)$, where D is the diameter of T and n^* is the size of the transitive closure of T.

Related Work. The concept of graph spanners was introduced in [13] where the authors used it to construct good synchronizers for communication networks. Peleg and Upfal [14] showed that using spanners as a subnet for routing helps in optimizing 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 with ratio $O(\log n)$ in unweighted graphs [4].

As mentioned before, the All-Best-Swaps 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 a MST, all the best swap edges can be found in $O(m)$ time [3]. When the original tree is a SPT and the objective is minimizing the (maximum or total) distance from a fixed root, the ABS problem has been solved in $O(n^2)$ time [11]. While these solutions are centralized, there also exist distributed solutions for computing best swaps in both the SPT [7] and the MST [6], where the efficiency is measured in terms of the communication cost in performing the computation. For minimum spanning trees, a stronger version of the problem was studied in [6] where the failures were assumed to occur at the nodes of the network, thereby disabling several edges at the same time.

In the case of minimum diameter spanning trees (MDST), there exists a centralized solution [8] that requires $O(m \log n)$ time and $O(m)$ space. The distributed version of the problem was solved in [9] using $O(\max\{n^*, m\})$ 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.

1. A communication network is a 2-edge-connected, undirected graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges. Each edge $e \in E$ has a non-negative real *length* $|e|$.

2. The length $|\mathcal{P}|$ of a path $\mathcal{P} = \langle p_1, \ldots, p_r \rangle$ is the sum of the lengths of its edges, and the *distance* $d_H(x, y)$ between any two vertices x, y in a graph H is the length of a shortest path in H between x and y.
3. The *stretch* of a spanning tree T of G is

$$\max_{x,y \in V} \{d_T(x, y)/d_G(x, y)\}$$

   An *optimal tree spanner* is a spanning tree with minimum stretch.
4. We often consider a spanning tree T to be rooted at some node u. For each node $x \neq u$, we then denote the *parent* of x by $p(x)$ and the set of its *children* by $C(x)$. Furthermore, let $T_x = (V(T_x), E(T_x))$ be the subtree of T rooted at x, including x.
5. The removal of any edge $e = (x, y)$ from T partitions the spanning tree into two disjoint trees T^x and T^y, where T^x contains node x and T^y contains y. A *swap edge* f for e is any edge in $E \backslash E_T$ that (re-)connects T^x and T^y, i.e., for which $T_{e/f} := (V, E_T \backslash \{e\} \cup \{f\})$ is a spanning tree of $G - e := (V, E \backslash \{e\})$. 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 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 $f \in E \backslash E_T$ is called a *candidate swap edge*, as it is a swap edge for at least one edge in T.
6. 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.

The following property simplifies the computation of the stretch for a given swap edge f with respect to a given failing edge e:

Observation 1 (follows from Lemma 16.1.1 in [12].) *Let $G = (V, E)$ be a weighted, undirected graph and let T be an optimal tree spanner of G with stretch k. Consider the spanning tree $T_{e/f}$ which results in deleting edge e from T and inserting f instead. Let $a, b \in V$ be two non-adjacent nodes in G, i.e., $(a, b) \notin E$, whose stretch in $T_{e/f}$ is $k' > k$. Then, there exists an edge $(x, y) \in E$ such that the stretch of the pair (x,y) in $T_{e/f}$ is at least k'.*

This observation motivates the following concepts, which we use for the description of our algorithms:

1. Each pair of nodes a, b with $(a, b) \in E$ is called a *stretch pair*. A stretch pair $g = (a, b)$ is *relevant* for measuring the stretch of any swap edge replacing a given failing edge e if the cycle which g forms with T contains the edge e (in other words, if g is also a swap edge for e).
2. For any edge $e \in T$ and any swap edge $f = (u, v) \in E \backslash E_T$ for e, the *stretch of a relevant stretch pair* (a, b) is the ratio of the distance between a and b in $T_{e/f}$, over the distance between them in G (i.e. $d_{T_{e/f}}(a, b)$ / $d_G(a, b)$). We can then express the stretch of $T_{e/f}$ as the maximum over the stretches of all the stretch pairs which are relevant for f replacing e.

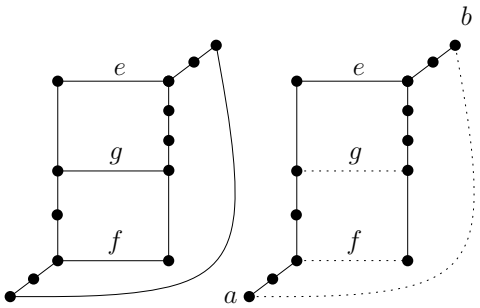

Fig. 1. An example showing that minimizing detour length does not minimize the stretch: On the left side, the 2-edge-connected graph G is shown, and on the right side the given tree spanner with stretch 8 is shown. Assuming that all edges have equal weight, the swap edge f minimizes the stretch to the 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, does not always give the optimal solution. For instance, see the counter-example shown in Figure 1 (This example can be generalized to obtain an arbitrary large difference between the stretch for the best swap f and the minimum detour edge g.)

The brute-force method for solving the *All-Best-Swaps* problem in a tree spanner, is to compute the stretch of $T_{e/f}$ for every edge pair (e, f) where $e \in T$ and f is a candidate swap edge for replacing edge e. There are $O(nm)$ such pairs and for each pair (e, f), the stretch of the tree $T_{e/f}$ can be computed in $O(m)$ time due to the following lemma.

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.*

The following preprocessing is required for the above computation. 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 pre-order traversal of T. Finally we construct a data structure which provides the nearest common ancestor 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 [10].)

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 have $d_{T_{e/f}}(u, v) = d_T(u, p) + |f| + d_T(q, v)$. where $f = (p, q)$ and p lies on u's side of the cut induced by e. In general, $d_T(u, p) = d_T(v, \text{nca}(u, p)) + d_T(p, \text{nca}(u, p))$. Both of these two terms can be computed as the absolute difference between the "to-root" distance of the two nodes involved. 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:

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 An $O(m^2 \log n)$ Time Solution for Weighted Graphs

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 $O(m + n)$ space.

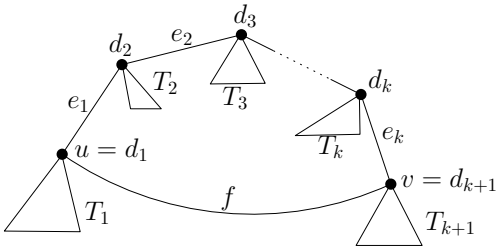

Fig. 2. The cycle that a non-tree edge f forms with the given spanning tree.

The idea of the algorithm (called *BestSwaps*) is sketched in 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. Note that all nodes of V which are not on the path in T from $u = d_1$ to $v = d_{k+1}$ lie in some subtree T_i of T which is rooted at node d_i. For a given failing edge $e_i = (d_i, d_{i+1})$ for which f is a relevant swap edge, the set of relevant stretch pairs contain all pairs(edges) where one endpoint lies in some tree attached to any of $d_1, \ldots, d_i$, and the other endpoint lies in some tree attached to any of $d_{i+1}, \ldots, d_{k+1}$. We associate with each node the index i of the subtree T_i containing it. For any edge $(a, b) \in E \setminus E_T$, this

[2] This can be checked using a "preorder/inverted preorder" labelling. For details, see e.g. [7], Section 3.2.

Algorithm 1: BestSwaps

```
Initialize Current-Best(e) for all e ∈ T
for f = (u, v) ∈ E\T do
    Relabel the vertices 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)
```

defines an order of the endpoints: we say that a is the *lower* endpoint if its index is smaller than the index of b. 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. Notice that irrespective of the failing edge, any relevant stretch pair for f, would be an edge $(a, b) \in E \setminus E_T$ where the endpoints a and b would have different indices. We maintain a data structure H described below which stores these relevant stretch pairs.

We consider the potential failing edges $e_1, e_2, \ldots, e_k$, in that order and evaluate f as potential best swap with respect to each e_i in turn. To that end, 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. We store the set $S(e_i)$ in our 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 priority queue (or heap) where priority is defined by the stretch value. 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 (we maintain these in a separate data structure) and update the current swap edge for e_i if required. Once we have repeated the above process for each edge $f \in E \setminus E_T$, we have obtained for each edge in T a best swap edge. We have:

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 $f = (u, v) \in G \setminus T$, the algorithm does the following. The nodes of the tree are re-labelled in $O(n)$ time. The non-tree edges of the graph G are partitioned into sets Start_i and End_i, corresponding to vertices d_i in the path from u to v. This can be done in $O(m)$ time. Further for each pair

(f, e) of swap edge f and failing edge e, the algorithm does some insertions and deletions[3] to the heap H. For any swap edge f and all possible failing edges e_i, any edge is inserted at most once and deleted at most once. Thus we require $O(m)$ heap operations, i.e. $O(m \log m)$ time corresponding to each such edge $f \in G \setminus T$. Note that we also need to compute the stretch of a stretch pair before inserting it. However this takes constant time using the techniques described in Section 2. Thus, overall the algorithm requires $O(m^2 \log m) = O(m^2 \log n)$ time.

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

4 An $O(n^3)$ Time Solution for Unweighted Graphs

In this section we consider a dynamic programming approach for computing all best swaps in an unweighted graph. We compute 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 (for "left") and T^r (for "right") of $T - e$ at the nodes l and r, respectively. Recall that by Observation 1, 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 $a, x \in T^l$ and $b, y \in T^r$ 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. That is, the length of a longest stretch path of a type i for a given node pair a, b can be computed in constant time, given only information that was previously computed. 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 $a \in T^l$ and $b \in T^r$ 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 incident to

[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 efficiently.

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

a (and hence does not contain any other edge from T^l), 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

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

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) for any $a, b \in V$ can be computed in $O(n^2)$ time.

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

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

Naturally, the symmetric equation holds for `Up-Stay`(a, b). Again, this recursion can be used for dynamic programming: all `Stay-Stay` and `Stay-Down` have been previously computed. In order to guarantee that `Stay-Up`$(a, p(b))$ is available, we consider the pairs a, b in an order where the b's occur in preorder. Thus, `Stay-Up`(a, b) and `Up-Stay`(a, b) for any $a, b \in V$ can also be computed in $O(n^2)$ time.

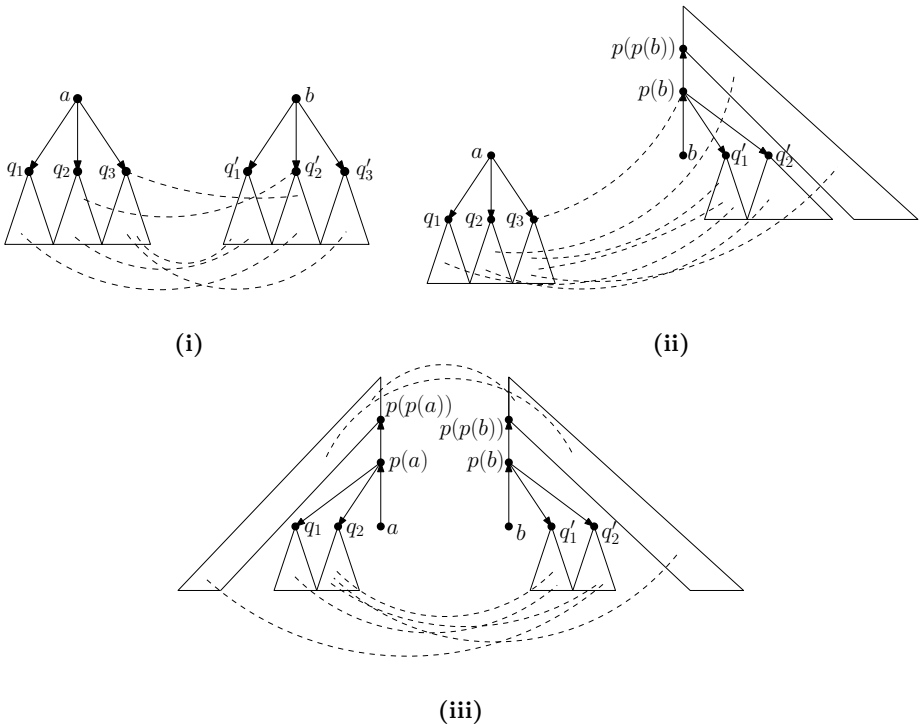

Fig. 3. Three of the possible stretch path types of the node pair a, b: (i) `Down-Down`, (ii) `Down-Up`, (iii) `Up-Up`

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 ordered such that both the a's and the b's occur in a postorder (note that this is easily possible). Hence, $\texttt{Down-Down}(a,b)$ for all $a, b \in V$ can be computed in $O(n^2)$ time.

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

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

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 $\texttt{Up-Down}(a, p(b))$ is obtained 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))

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

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 $O(n)$ edges $e \in E_T$. Then, the algorithm computes, for each swap edge candidate $f = (u, v)$, the stretch of $T_{e/f}$ as

$$\max\Big\{ \begin{array}{lll} \texttt{Stay-Stay}(u, v), & \texttt{Stay-Down}(u, v), & \texttt{Down-Stay}(u, v), \\ \texttt{Stay-Up}(u, v), & \texttt{Up-Stay}(u, v), & \texttt{Down-Down}(u, v), \\ \texttt{Down-Up}(u, v), & \texttt{Up-Down}(u, v), & \texttt{Up-Up}(u, v) \end{array} \Big\},$$

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 $O(n^2)$. In short, we have the following:

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 to All-Best-Swaps for Spanners

In this section, we consider the scenario when each node in the network has only local information about the network. We are interested in a distributed algorithm which enables each node to compute the information required by it 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 incident edges. Note that the information about the best swap edge for any particular edge $e \in T$, needs to be stored only at the endpoints of e [9]. For the following, let n^* denote the size of the transitive closure of the tree T, when the edges are directed towards the root node.

We make the following assumptions:

- The tree T is rooted at fixed node r, which is know to every node $x \in G$.
- Each node $x \in G$ knows which of its incident edges connect to its children in T and which edge connects to its parent in T.
- Each node x knows the weights of all edges in G that are incident to it.
- Each node x knows the distance $d_T(x, v)$ to any node $v \in T$. (Note that this can be easily pre-computed using $O(n^*)$ messages.)
- The nodes of the tree T are labelled in such a way that given the labels of two nodes $a, b \in T$, it is possible to find the label of the nearest common ancestor of a and b (i.e. $nca(a, b)$). This can be done using labels of size $O(\log n)$ as shown in [1].

Given a rooted spanning tree T of G and any node $v \in T$, we define T_v as the subtree of T that is rooted at v and we denote by $Start(T, v)$ the set of all non-tree edges whose one end-point is in T_v and the other end-point is outside T_v. At each node x and for each edge $e = (x, y) \in T$, we wish to compute a list of all swap edges with one end-point in T^x and the other endpoint in T^y. If x is a child of y, then this list is same as $Start(T, x)$. Otherwise this list is same as $Start(T, y)$.

The idea of our algorithm is to compute all these lists using the converge-cast technique on the tree, once propagating information from the leaves of T up to the root and then from the root back to the leaves again. Notice that each leaf node already has the required information.

From the above computation, node x has a list of all potential swap edges connecting T_x to T_y (along with their weights). Given any two such edges $f = (u, v)$ and $g = (a, b)$, it is possible to compute the stretch $d_{T_{e/f}}(a, b)$ as follows:

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

$d_T(a, u)$ can be computed as $d_T(a, u) = d_T(a, x) + d_T(x, u) - 2 \cdot d_T(x, nca(a, u))$. Similarly $d_T(v, b)$ can also be computed. Thus node x can compute the stretch of every swap edge and thus determine the best swap edge for e.

Complexity of the Algorithm. The above algorithm requires only $O(D)$ time for trees of diameter D (measured as the number of "hops"), because the message from the farthest leaf has to reach the root and vice versa. Note that the only information exchanged between the nodes is the list of candidate swap edges. Information about each swap edge $f = (u, v)$ travels only along the path in T from u to v. So the overall communication complexity is $O(n^*)$ times the size of node labels (and edge weights).

6 Swapping versus Recomputing a New Tree Spanner

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. (Recall that the stretch for both trees is computed with respect to G, and not $G - e$.)

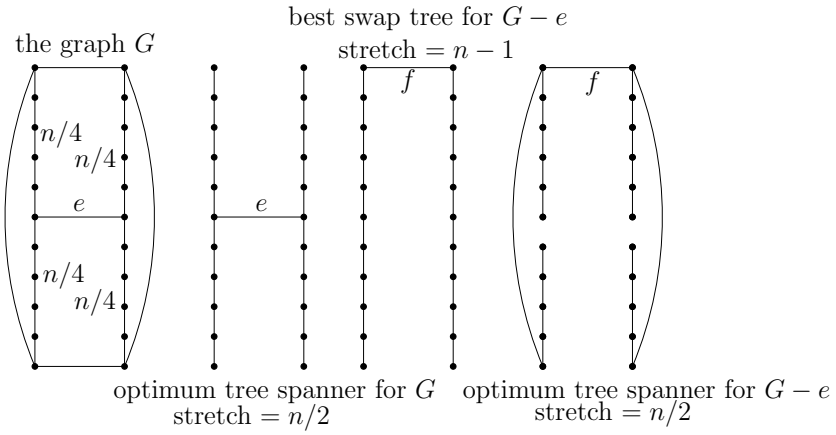

Fig. 4. An example of a graph where the stretch of the best swap tree is 2 times worse than the best newly computed tree spanner.

Lemma 2. *For any failing edge e of an optimal tree spanner of an* unweighted *graph G, the maximum stretch of the* swap tree *is at most* 2 *times larger than the stretch of an optimal tree spanner of $G - e$. The bound of* 2 *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 fails. Let (a, b) be the pair of nodes 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 best swap edge for e. We have

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

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 2 is shown in Figure 4. □

References

[1] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and a new algorithm for a distributed environment. *Theory Comput. Syst.*, 37(3):441–456, 2004.

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

[3] 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.

[4] 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, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[5] 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.

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

[7] P. Flocchini, L. Pagli, G. Prencipe, N. Santoro, and P. Widmayer. Computing All the Best Swap Edges Distributively. In *Proceedings of Symposium on Principles of Distributed Systems (OPODIS)*, volume 3544 of *LNCS*, pages 154–168, 2004.

[8] B. Gfeller. Faster Swap Edge Computation in Minimum Diameter Spanning Trees. In *16th Annual European Symposium on Algorithms (ESA'08)*, 2008. To appear.

[9] B. Gfeller, N. Santoro, and P. Widmayer. A Distributed Algorithm for Finding All Best Swap Edges of a Minimum Diameter Spanning Tree. In A. Pelc, editor, *DISC*, volume 4731 of *LNCS*, pages 268–282. Springer, 2007.

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

[11] E. Nardelli, G. Proietti, and P. Widmayer. How to Swap a Failing Edge of a Single Source Shortest Paths Tree. In *COCOON*, pages 144–153, 1999.

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

[13] 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.

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