PhD Thesis on

Fault-tolerant networks: fast and effective solutions in centralized and distributed settings

PhD Program in Computer Science: XXX cycle

Supervisor:

Prof. Guido Proietti
guido.proietti@univaq.it

Tutor:

Dr. Mattia D'Emidio
mattia.demidio@gssi.it

Author:

Feliciano Colella
feliciano.colella@gssi.it

The author declares that the content of this thesis has been published in co-authored papers in conference proceedings.

In particular, the content showed in Chapter 3 has been published in the Proceedings of the "22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2015)" and the "24th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2017)".

Moreover, the content in Chapter 4 has been accepted to the "28th International Symposium on Algorithms and Computation (ISAAC 2017)" and will be presented in December, 2017.

Finally, the work presented in Part 2 has been published in the Proceedings of the "18th Italian Conference on Theoretical Computer Science (ICTCS 2017)".

ABSTRACT

Nowadays there is an increasing demand for an efficient and resilient information exchange in communication networks. This means to design, on the one hand, a logical structure onto a given communication infrastructure, which optimizes some sought routing protocol in the absence of failures. On the other hand, to make such a structure resistant against possible link/node malfunctioning, by either adding to it a set of redundant links, which will enter into operation as soon as a failure takes place, or by modifying the existing protocols in order to be able to deal with any malfunctions. Therefore, in the recent past, a lot of work has been done towards the designing of network structures and protocols as reliable as possible. Many different approaches and solutions has been devised, in a broad assortment of settings. The aim of this thesis is to tackle a couple of this settings and solve some new problems and improve some existing results. More in detail we focus on a particular class of problems arising in the centralized setting, namely, the *all-best-swap-edge* problems. In this case, we want to be able to find a substitute for any link in a *tree-based* network, that will enter into operation after any malfunction of the original link. Furthermore, we show a result on a problem that belong to the converse setting, the distributed one. In particular, we propose an approximate fault-tolerant path-reporting *labeling scheme*, based on the so-called 2-Hop Cover strategy. In this case, we want to be able to efficiently find a path within a network, even in the presence of a certain set of malfunctions, without using a centralized computation.

Acknowledgments

I would like to thank my advisor, Prof. Guido Proietti, and all my co-authors Davide Bilò, Mattia D'Emidio, Luciano Gualà and Stefano Leucci for all the help and useful experience gained from them. I am grateful to my parents for supporting me during these three years and I would like to thank all my colleagues at the GSSI who have made my experience way more enjoyable. Finally, I would like to thank all of my friends that have always been near me during this journey.

Contents

CHAPTER 1

Introduction

1.1. Motivation

Due to the large size of the recent networks, and the huge amount of information and messages exchanged, there is an increasing demand for an efficient and resilient information exchange in communication networks. Overcoming this problem means, on the one hand, to design a logical structure onto a given communication infrastructure, which optimizes some sought routing protocol in the absence of failures and, on the other hand, to make such a structure resistant against possible link/node malfunctioning. In this case routing protocols/algorithms must be able to efficiently route a message from a source towards a destination. Doing so in presence of failures is obviously more difficult and, therefore, appropriate solutions must be devised. This can be achieved in many ways, for example by suitably adding to it a set of redundant links, which will enter into operation as soon as a failure takes place, or modifying the existing protocols in order to be able to deal with any malfunctions. Unfortunately, it can be shown, as we will discuss in the following, that pursuing optimality might require the activation of a large number of edges. This would occur even if we only consider edge-failures in the *single-source* case in which we are interested in preserving the shortest paths emanating from a single vertex. Asymptotically speaking, the number of edges needed can coincide with the number of available links in the infrastructure even when such an infrastructure is complete, i.e., it contains all the point-to-point links between hosts.

In fact, in the recent past, the problem of designing *sparse* fault-tolerant structures and protocols has received much attention. Many different approaches and solutions have been devised, in a broad assortment of settings.

In this thesis we consider some prominent problems and we provide both solutions to natural problems as well as improving existing results on established models. In particular we consider the following settings.

Computing swap edge: We focus on a particular class of problems arising in the centralized setting, namely, the all-best-swap-edge problems. In this setting, the underlying network is represented with a tree, therefore it is really susceptible to failures. In fact, a single failing link will split the network into two non-communicating sub-networks. Therefore, we want to be able to find a substitute for any link in the network that can fail. Such a link will be swapped with the failing one, in front of any malfunction. Moreover we would also like to optimize some objective function, related with the underlying tree network. Two different cases will be investigated, in the first the underlying tree network will be a *tree spanner*, while in the second case it will be a more classic *single-source shortest path tree*.

Labeling scheme: We consider also a problem that belongs on the orthogonal setting, the distributed one. In particular, we develop an approximate fault-tolerant path-reporting labeling scheme, based on the 2-Hop cover strategy. A labeling scheme is a method to retrieve some global property of a graph by exploiting (in a distributed fashion) the information stored, in the form of (short) labels, at its vertices. The 2-Hop cover strategy is based on the idea of representing the shortest paths of a graph as the concatenation at an intermediate so-called hub vertex of the two shortest paths emanating from the corresponding end vertices. In our case, we want to be able to efficiently find a path within a network, even in presence of a certain set of edge failure, by means of a distributed re-routing computation. Note that our work contains a deep experimental analysis to validate the theoretical results.

1.2. Structure of the Thesis

The thesis is organized in two parts. Part 1 is concerned with swapping problems, where we propose different solutions to a set of problems in this scenario. In particular, Chapter 2 will thoroughly show the state of the art and the related works on swapping problems. Chapter 3 and 4 will contain the main results. Part 2 instead, contains an efficient approach to build a labeling scheme in the fault-tolerant setting. More in details, Chapter 6 will be an introduction to the problem and the related results, while Chapter 7 will contain all the technical details, as well as the experimental analysis.

Part 1

Swapping Problems

CHAPTER 2

Introduction

The problem of computing *all the best swap edges* (ABSE) of a tree has a long and rich algorithmic tradition. Basically, let $G = (V(G), E(G), w)$ be an n-vertex and m-edge 2-edge-connected undirected graph, with edge-weight function $w : E(G) \rightarrow \mathbb{R}^+$, and assume we are given a spanning tree T of G, which was computed by addressing some criterion (i.e., objective function) ϕ. Then, the problem is that of computing a *best swap edge* (BSE) for every edge $e \in E(T)$, namely an edge $f \in E(G) \setminus E(T)$ such that the swap tree $T_{e/f}$ obtained by swapping e with f in T optimizes some objective function ϕ' out of all possible swap trees. Quite reasonably, the function ϕ' must be related (if not coinciding at all) with ϕ.

The first immediate motivation for studying an ABSE problem comes from the edge fault-tolerance setting – a commonly accepted framework. Broadly speaking, the algorithmic question here is to design *sparse* subgraphs that guarantee a proper level of functionality even in the presence of an edge failure. In such a context, the rationale of an ABSE-based solution is the following: operations are normally performed on a (possibly optimal) spanning tree, and whenever an edge failure takes place, a corresponding BSE is plugged in. This way, the connectivity is reestablished in the most prompt and effective possible way (see also [41, 50] for some additional practical motivations).

Besides their practical relevance, ABSE problems have also an interesting theoretical motivation. Indeed, swapping can be reviewed as an exploration of the space of the perturbed (w.r.t. an edge swap) solutions to a given spanning tree optimization problem. Thus, the algorithmic challenge of solving efficiently an ABSE problem is related with the understanding of the structure of this space of perturbed solutions. Sometimes a perturbed solution is guaranteed to be optimal w.r.t. the optimization function originally addressed

by the spanning tree, some other time the quality of the swap tree is far apart from that of a new optimal solution. This is exactly why each ABSE problem has its own combinatorial richness, and thus requires a specific approach to be solved efficiently. Then, different ABSE problems have required the use of completely different approaches and methods in order to obtain efficient solutions. Therefore, in the vast zoo of spanning tree optimization problems, the efficiency of swapping, once we analyze it for each and every tree edge, helps to understand the structural robustness of a spanning tree solution.

In this part of the thesis, we solve the ABSE problem in two different cases. In the first one, we solve the ABSE problem when the tree is a tree spanner. In the second case, we focus onto two different ABSE problems on a shortest path tree.

2.1. Related Work

2.1.1. Swapping on a Tree Spanner.

One of the results presented in this thesis, regards an ABSE problem on an elusive spanning tree structure, namely the *tree spanner* (ABSE-TS problem in the following). A tree spanner is built with the aim of preserving node-to-node distances in G. Indeed, the *stretch factor* σ of a spanning tree T of G is defined as the *maximum*, over all the pairs $u, v \in V(G)$, of $d_T(u, v)/d_G(u, v)$, where d_T and d_G denote the distance (i.e., the length of a shortest path) in T and G, respectively. Correspondingly, an *optimal* tree spanner has minimum stretch out of all the spanning trees of G. Unfortunately, embedding a graph into a tree spanner is a hard task: on the one hand, the input graph might be such that no any spanning tree provides a good preservation of the original distances (just think to the case in which G is an unweighted cycle), and on the other hand finding an optimal tree spanner is notoriously an APX-hard problem, with no known $o(n)$-approximation. Hence, once a given solution undergoes a transient edge failure, the recomputation from scratch of a new (near) optimal solution is computationally unfeasible. Moreover, as for other distance-based structures, a tree spanner is highly sensitive to a perturbation of the input instance, and already a single edge removal might completely alter

the topology of the solution at hand (i.e., in practical terms, this will require the set up of many new links and a large adjustment of routing tables).

Thus, swapping in a tree spanner is even more attractive than in general, since it allows for drastically reducing the foreseen computational and set-up costs. Indeed, the ABSE-TS problem was studied in [26], where the authors devised two solutions for both the weighted and the unweighted case, running in $O(m^2 \log n)$ and $O(n^3)$ time, respectively, and using $O(m)$ and $O(n^2)$ space, respectively. However, there the authors assume that a BSE is an edge minimizing the stretch of the swap tree w.r.t. distances in the *original* graph G, and not in the graph G deprived of e, say $G - e$. This contrasts with the general assumption (and the intuition) that the quality of a swap tree should be evaluated in the surviving graph, and not in the original graph. Hence, in [5] the authors resorted to such a standard setting, and provided two efficient linear-space solutions for both the weighted and the unweighted case, running in $O(m^2 \log \alpha(m, n))$ (where α denotes the inverse of the Ackermann function) and $O(mn \log n)$ time, respectively. Notice that from a computational point of view, as shown in [5], the two settings are substantially equivalent, so our solutions can be used to improve the results given in [26] as well.

Besides swap problems, the research on tree spanners is very active, also due to the strong relationship with the huge literature on *spanners*, where distances in G are approximately preserved through a *sparse* spanning subgraph. As mentioned before, finding an optimal tree spanner is a quite hard problem. More precisely, on weighted graphs, if G does not admit a tree 1-spanner (i.e., a spanning tree with $\sigma = 1$, which can be established in polynomial time [15]), then the problem is not approximable within any constant factor better than 2, unless P=NP [44]. In terms of approximability, no non-trivial upper bounds are known, except for the $O(n)$-approximation factor returned by a *minimum spanning tree* (MST) of G. If G is *unweighted*, things go slightly better. More precisely, in this case the problem becomes $O(\log n)$-approximable, while unless P=NP, the problem is not approximable within an additive term of $o(n)$ [31]. Moreover, the corresponding decision problem of establishing whether G admits a tree spanner with stretch σ is NP-complete for every fixed $\sigma \geq 4$ (for

$\sigma = 2$ it is polynomial-time solvable [15], while for $\sigma = 3$ the problem is open). Finally, it is known that constant-stretch tree spanners can be found for several special classes of (unweighted) graphs, like strongly chordal, interval, and permutation graphs (see [13] and the references therein).

For the sake of completeness, we mention that for the related concept of *average* tree σ-spanners, where the focus is on the average stretch w.r.t. all node-to-node distances, it was shown that every graph admits an average tree $O(1)$-spanner [1].

To conclude, we point out that the general problem of designing *fault-tolerant spanners* has been extensively studied in the literature, and we refer the interested reader to [16, 30, 7] and the references therein.

2.1.2. Swapping on a Shortest Path Tree.

Another interesting ABSE problem studied in this thesis is the one where the underlying tree is a *Shortest Path Tree* (SPT), namely a tree containing the union of all the shortest paths emanating from a given distinguished source node. Swapping in a SPT is a well-recognized approach, since it was shown already in [42] that an effective broadcast routing protocol can be put in place just after the original SPT undergoes an edge failure. Since an SPT enjoys several optimality criteria when looking at distances from the source, say s, several papers have analyzed the problem in various respects. However, most of the efforts focused on the minimization w.r.t. the following two swap criterion: the *maximum* and the *average* distance from s to any node which remained disconnected from s after a failure. Previously to the work developed in this thesis, the fastest solutions for these two ABSE problems run in $O(m \log \alpha(m, n))$ time [12] and $O(m\, \alpha(n, n) \log^2 n)$ time [29], respectively. On the other hand, concerning the *quality* of the swap, i.e., the ratio, in the worst case, between the selected criterion in the swap tree as opposed to that of a true SPT in the damaged graph, it is quite good in both cases, since it has been shown that in the swap tree the maximum (resp., average) distance of the disconnected nodes from s is at most twice (resp., triple) that of the new optimum SPT [47], and these bounds are tight.

Besides swap-based approaches, an SPT can be made edge-fault-tolerant by further enriching the set of additional edges, so that the obtained structure has almost-shortest paths emanating from the source, once an edge fails. The currently best trade off between the size of the set of additional edges and the quality of the resulting paths emanating from s is provided in [8], where the authors showed that for any arbitrary constant $\varepsilon > 0$, one can compute in polynomial time a slightly superlinear (in n, and depending on ε) number of additional edges in such a way that the resulting structure retains $(1+\varepsilon)$-stretched post-failure paths from the source.

2.1.3. Other Swapping Problems.

The most famous and studied ABSE problem comes when T is a *minimum spanning tree* (MST) of G. In this case, a best swap is of course a swap edge minimizing the *cost* (i.e., sum of the edge weights) of the swap tree, i.e., a swap edge of minimum weight (and we know this produces a MST of the perturbed graph). This problem is also known as the MST *sensitivity analysis* problem, and can be solved in $O(m \log \alpha(m,n))$ time [49], by using an efficient data structure, namely the *split-findmin* [35]. This was improving on another efficient solution given by Tarjan [54], running in $O(m\alpha(m,n))$ time and making use of the *transmuter*, namely a compact way of representing the cycles of a graph. Other data structures which revealed their usefulness to solve efficiently ABSE problems include *kinetic heaps* [12], *top trees* [5], *mergeable heaps* [48], and many others.

Concerning the SPT, other interesting swap criteria which have been analyzed include the minimization of the maximum increase (before and after the failure) of the distance from s, and the minimization of the distance from s to the root of the subtree that gets disconnected after the failure [46]. Besides the centralized setting, all these swap problems have been studied also in a distributed framework (e.g., see [32, 33, 34, 27]).

Concerning the problem of swapping in other spanning trees, this has received a significant attention from the algorithmic community. There is indeed a line of papers that address ABSE problems starting from different types of spanning trees. Just to mention a few, besides the MST, we recall the *minimum*

diameter spanning tree (MDST) and the *minimum routing-cost spanning tree* (MRCST). Concerning the MDST, a best swap is an edge minimizing the *diameter* of the swap tree [40, 46], and the best solution runs in $O(m \log \alpha(m,n))$ time [12]. Finally, regarding the MRCST, a best swap edge is clearly one minimizing the *all-to-all routing cost* of the swap tree [56], and the fastest solution for solving this problem has a running time of $O\left(m2^{O(\alpha(n,n))} \log^2 n\right)$ [11].

2.2. Our Results

In this part of the thesis, our results develop towards two different problems. First, we focus on the tree spanner, and we present a new algorithm that solves the ABSE-TS problem in $O(n^2 \log^4 n)$ time and $O(n^2 + m \log^2 n)$ space. Thus, our solution improves on both the algorithms that were provided in [5], for weighted and unweighted graphs, respectively, whenever $m = \Omega(n \log^3 n)$. Most remarkably, for dense weighted graphs, the improvement is almost quadratic in n. Second, we focus on the SPT and we tackle two new criteria for swapping in a SPT, namely the minimization of either the *maximum* or the *average stretch factor* from the root (ABSE-MS and ABSE-TS problems in the following), for which we propose two efficient solutions running in $O(mn+n^2 \log n)$ and $O(mn \log \alpha(m,n))$ time, respectively.

Before our work, no results were known for these cases, and this is very surprising, since they are (especially the former one) the universally accepted criterion leading to the design of ordinary spanners.

2.2.1. Swapping on a Tree Spanner.

Concerning the ABSE-TS problem, it is worth noticing that, as observed in [26], the estimation of the stretch of the swap tree induced by a *single* swap edge f for a given failing edge e, would in principle ask for the evaluation of the stretch of $O(m)$ relevant pairs of nodes in G, namely the endvertices of all the non-tree edges that may serve as swap edge for e besides f. In fact, a *critical edge* for f is the one whose endvertices maximize such a stretch out of these non-tree edges, and two swap edges will be essentially compared on the basis of their stretch w.r.t. their critical edge. This is basically the reason why both previous approaches take $\Omega(m^2)$ time. Thus, to avoid such a bottleneck, we

drastically reduce, on the one hand, the number of candidate best swap edges, and on the other hand, the number of potential critical edges that need to be checked. More precisely, for each of the $n-1$ considered edges in T, we succeed in reducing to $O(n \log n)$ the number of best swap edge candidates, and for each one of them we just need to check $O(\log^2 n)$ possible critical edges. The key ingredients to reach such a goal are the following:

Centroid decomposition.
A *centroid decomposition* of T, which consists of a log-depth hierarchical decomposition of the vertices in T; a careful use of such a decomposition, combined with a set of preprocessing steps that associate various information with the tree nodes, allows us to reduce the number of candidate BSEs and of their corresponding candidate critical edges. As far as we know, this is the first time that such a decomposition is used to solve an ABSE problem, and we believe it will possibly be useful in other contexts as well.

Upper envelope.
The second ingredient is given by the dynamic maintenance of the *upper envelopes* of a set of linear functions. Each of these functions is associated with a non-tree edge, and whenever the failure of a given tree edge is considered, it expresses the stretch such a non-tree edge induces w.r.t. a variable candidate BSE. This way, when we have to find a critical edge for a given candidate BSE f, we have to select the *maximum* out of all the functions once they are evaluated in f. In geometric terms, this translates into the maintenance of the upper envelope of a set of functions, with the additional complication that, for consistency reasons, this set of functions must be suitably partitioned into groups according to the underlying centroid decomposition, and moreover these groups are dynamic, since they depend on the currently considered tree edge.

2.2.2. Swapping on a Shortest Path Tree.

Concerning the `ABSE-MS` and the `ABSE-AS` problems, first of all notice that both our solutions incorporate the running time for computing all the replacement

shortest paths from the source after the failure of every edge of the SPT, which takes $O(mn \log \alpha(m,n))$ time as shown in [36], whose computation essentially dominates in an asymptotic sense the time complexity.[1] Our results are based on independent ideas, as described in the following.

Maximum stretch problem (`ABSE-MS`).
For the `ABSE-MS` problem, we develop a *centroid decomposition* of the SPT, and we exploit a distance property that has to be enjoyed by a BSE w.r.t. a nested and log-depth hierarchy of centroids, which will be defined by the subtree detached from the source after the currently analyzed edge failure. A further simple filtering trick on the set of potential swap edges will allow to reduce them from $O(m)$ to $O(n)$, thus returning the promised $O(n^2 \log n)$ time.

Average stretch problem (`ABSE-AS`).
For the `ABSE-AS` problem, we instead suitably combine a set of information that are linearly-computable at every edge fault. Such information will essentially allow to describe in $O(1)$ time the quality of a swap edge. This procedure is in principle not obvious, since to compute the average stretch we need to know, for each swap edge, the $O(n)$ distances to all the nodes in the detached subtree. Again, by filtering on the set of potential swap edges, we will get an $O(n^2)$ running time, which will be absorbed by the all-replacement paths time complexity.

Quality Analysis.
Concerning the quality of the corresponding swap trees, we instead show that the guaranteed (either maximum or average, respectively) stretch factor w.r.t. the paths emanating from the source (in the surviving graph) is equal to 3, and this is tight. By using a different terminology, our structures can then be revised as *edge-fault-tolerant single-source* 3*-spanners*, and we qualified them as *effective* since they can be computed quickly, are very sparse, provide a very

[1] To be more precise, what is needed, by definition, are all the replacement *distances*, but to the best of our knowledge no faster solution (for instance, based on a min-plus matrix multiplier) is known in order to find them.

simple alternative post-failure routing, and finally have a small (either maximum or average) stretch.

CHAPTER 3

Swapping on a Tree Spanner

3.1. Weighted case

3.1.1. Preliminary definitions.

Let $G = (V(G), E(G), w)$ be a 2-edge-connected, edge-weighted, and undirected graph with cost function $w : E(G) \to \mathbb{R}^+$. We denote by n and m the number of vertices and edges of G, respectively. If $X \subseteq V(G)$, let $E(X)$ be the set of edges incident to at least one vertex in X. When $X = \{v\}$, we may write $E(v)$ instead of $E(\{v\})$. Given an edge $e \in E(G)$, we will denote by $G - e$ the graph obtained from G by removing edge e. Similarly, given a vertex $v \in V(G)$, we will denote by $G - v$ the graph obtained from G by removing vertex v and all its incident edges.

Given an edge $e \in E(T)$, we let $S(e)$ be the set of all the *swap edges* for e, i.e., all edges in $E(G) \setminus \{e\}$ whose endpoints lie in two different connected components of $T - e$. We also define $S(e, X) = S(e) \cap E(X)$, and $S(e, X, Y) = S(e) \cap E(X) \cap E(Y)$. When $X = \{v\}$, we will simply write $S(e, v)$ in lieu of $S(e, \{v\})$. For any $e \in E(T)$ and $f \in S(e)$, let $T_{e/f}$ denote the *swap tree* obtained from T by replacing e with f.

Given two vertices $x, y \in V(G)$, we denote by $d_G(x, y)$ the *distance* between x and y in G. We define the *stretch factor* of the pair (x, y) w.r.t. G and T as $\sigma_G(T, x, y) = \frac{d_T(x,y)}{d_G(x,y)}$. Accordingly, the stretch factor $\sigma_G(T)$ of T w.r.t. G is defined as $\sigma_G(T) = \max_{x,y \in V(G)} \sigma_G(T, x, y)$.

DEFINITION 1 (Best Swap Edge). *An edge $f^* \in S(e)$ is a best swap edge (BSE) for e if $f^* \in \arg\min_{f \in S(e)} \sigma_{G-e}(T_{e/f})$.*

In the sequel, in order to solve the ABSE-TS problem, we will show how to efficiently find a BSE for every edge e of a tree spanner T of G.

3.1.2. High-level description of the algorithm.

At a very high level, our algorithm works as follows. Initially, we root T at an arbitrary vertex, and we consider one after the other in a post-order fashion (the failure of) all the edges of T. Once that an edge e is considered, we focus on the root's subtree (as induced by the removal of e), and for each vertex v of such subtree, we select in $O(\log^4 n)$ time a best possible swap edge incident to v. This subroutine is exactly the core step of our approach, and its runtime is obtained through a non-trivial mixing of new techniques and sophisticate data structures. More precisely, a key role in our approach is played by an initial *centroid decomposition* of T, combined with a set of preprocessing steps that associate several information with the tree nodes. As we will show in the following chapters, the efficiency of this approach stands in the ability of selecting in poly-logarithmic time a *critical edge*, say $g = (x, y)$, associated with a given swap edge incident in v, say $f = (v, z)$. In other words, the endvertices of g are exactly the pair of nodes in G for which the stretch of f in $T_{e/f}$ is maximized.

To find g efficiently, we strongly make use of the log-depth of the centroid decomposition, and moreover we represent the reciprocal level of criticality of two competing swap edges through a function which is independent of e. And exactly this independence enables us to manage all these functions (suitably partitioned on the basis of the centroid decomposition of T), and eventually to select a BSE, via the dynamic maintenance of their *upper envelopes*.

It is useful to consider the tree T as rooted at any fixed vertex, and to assume, w.l.o.g., that T is binary. Indeed, if T is not binary, then it is possible, by using standard techniques, to transform G and T into an equivalent graph G' and a corresponding binary spanning tree T', with $|V(G')| = \Theta(n)$ and $|E(G')| = \Theta(m)$, and such that a BSE for any edge of T is univocally associated with a BSE for a corresponding edge of T'. This transformation requires linear time and it is sketched in the following subsection.

3.1.3. Reducing the degree of T.

Here we show a linear time reduction that transforms G and a rooted spanning

tree T of G into an equivalent graph $G' = (V(G'), E(G'), \overline{w})$ and a corresponding binary spanning tree T' such that $|V(G')| = \Theta(n)$ and $|E(G')| = \Theta(m)$.

Initially, G', w', and T' coincide with G, w, and T, respectively. We iteratively search for a vertex u in T' that has 3 or more children, and we lower its degree. Let $v_1, \ldots, v_h$, with $h \geq 3$, be the children of u. We remove all the edges $\{(u, v_i) : 1 \leq i \leq h\}$ from both G' and T', then we add to both G' and T' a binary tree whose root coincides with u, and that has exactly h leaves $x_1, \ldots, x_h$. We assign weight $w'(e) = 0$ to all the edges e of this tree. Finally, we add to G' and T' an edge (x_i, v_i) for each $1 \leq i \leq h$, and we set $w'(x_i, v_i) = w(u, v_i)$. An example of such a transformation is shown in Figure 1.

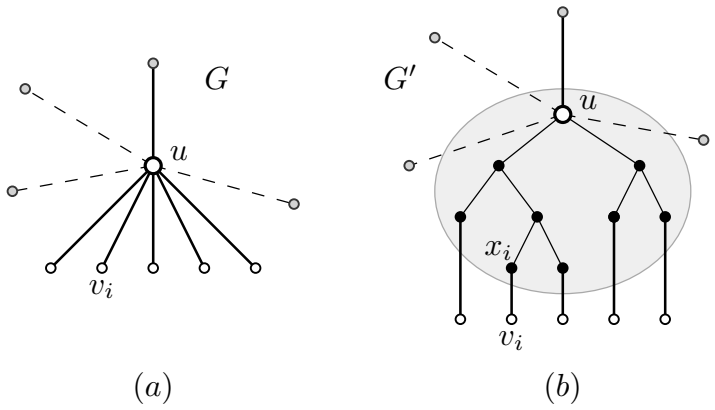

FIGURE 1. Reducing the degree of vertices in G: on the left side, the tree T (solid edges) embedded in G, on the right side the superimposition of the binary tree to T in order to get a maximum degree of 3. Solid edges in the gray area have weight 0, while the weight of (x_i, v_i) is $w(u, v_i)$.

Clearly, it holds that $|V(G')| = O(|V(G)|)$ and $|E(G')| = O(|E(G)|)$. Moreover, the computation of G' and T' requires linear time. Now, observe that, for every $a, b \in V(G)$, it holds that $d_{T_{e/f}}(a, b) = d_{T'_{e/f}}(a, b)$. Furthermore, for every edge $e = (u, v_i)$ of T, f is a swap edge for e in T if and only if f is a swap edge for the edge (z, v_i) in T', where z is the parent of v_i in T'. As a consequence, we can conclude that, for every edge $e = (u, v_i)$ of T, $f \in S(e)$ is a BSE for e w.r.t. T if and only if f is a BSE for the edge (z, v_i) w.r.t. T', where z is the parent of v_i in T'.

3.1.4. Centroid Decomposition.

As a preprocessing step, we compute a *centroid decomposition* of T. A *centroid* of an n-vertex tree is a vertex whose removal splits T into subtrees of size at most $n/2$ [43]. A centroid decomposition of T can be computed in $O(n \log n)$ time, and can be represented by a tree $\mathcal{T}$ of height $O(\log n)$, whose nodes are actually subtrees of T. $\mathcal{T}$ is recursively defined as follows: the root of $\mathcal{T}$ is T. Then, let τ be a node of $\mathcal{T}$ (i.e., a subtree of T) such that τ contains more than one vertex, and let c be a centroid of τ. Since T is binary, the forest $\tau - c$ contains at most 3 trees, that we call τ_c^1, τ_c^2, and τ_c^3 (if $\tau - c$ generates less than 3 subtrees, we allow some τ_c^i to be the empty tree). Moreover, let τ_c^0 be the subtree of T containing the sole vertex c. Then, τ will have in $\mathcal{T}$ a child for each of the subtrees $\tau_c^i, i = 0, \ldots, 3$ (see Figure 2 (a)). Since a centroid on a n-vertex tree can be found in linear time, the whole procedure requires $O(n \log n)$ time, and it is easy to see that the height of $\mathcal{T}$ is $O(\log n)$.

Our solution (see also the next algorithm box) works in $n - 1$ phases. In each phase, we consider a single failing edge $e \in E(T)$ and we look for a best swap edge for e. We process the edges in $E(T)$ in a bottom-up fashion so that, when edge e is considered, all the edges in $E(T_e)$ have already been processed. Let $e \in E(T)$ be the currently considered edge, and let U_e (resp. D_e) be the set of vertices that belong to the connected component of $T - e$ that contains (resp. does not contain) the root of T. We break down each of these phases into $O(n)$ additional sub-phases: when edge e is failing, we consider all the vertices in U_e and, for each such vertex v, we solve a restricted version of the ABSE-TS problem where we compute: (i) a *v-restricted best swap edge* (v-BSE for short), i.e., an edge $f \in \arg\min_{f \in S(e,v)} \sigma_{G-e}(T_{e/f})$, and (ii) the corresponding stretch factor $\sigma_{G-e}(T_{e/f})$. To simplify handling of special cases, whenever $S(e, v) = \emptyset$, we assume that $f = \perp$ and that $\sigma_{G-e}(T_{e/f}) = +\infty$. As we will see in the rest of the chapter, the core of our algorithm is exactly the efficient computation of these v-BSEs and of their stretch factors. This is done trough a clever selection of a small set of *candidate* v-BSEs, as we will discuss in more detail in the next section. Once all the v-BSEs for e are computed, a BSE for e can be found as the one minimizing the associated stretch factor.

Procedure ABSE-TS(G, T)

$\mathcal{T} \leftarrow$ Centroid decomposition of T;
foreach $e \in E(T)$ *in postorder* **do** // $n-1$ phases
  $U_e \leftarrow$ vertices of the component of $T - e$ that contains the root of T;
  $f^* \leftarrow \perp$; // Current BSE for e
  foreach $v \in U_e$ **do** // $O(n)$ sub-phases
    compute a v-BSE f for e and the corresponding stretch factor;
    // This takes $O(\log^4 n)$ time by using $\mathcal{T}$ and the dynamic maintenance of the upper envelopes associated with the swap edges, as shown in Section 3.1.5
    if $\sigma_{G-e}(T_{e/f}) < \sigma_{G-e}(T_{e/f^*})$ **then** $f^* \leftarrow f$;
  return f^* as BSE for e and **continue** with the next phase.

3.1.5. Computing efficiently a v-BSE.

To show how a v-BSE for e can be computed efficiently, we need some preliminary definitions:

DEFINITION 2 (Critical Edge). *Given $e \in E(T)$ and a swap edge $f = (v, u) \in S(e, v)$, a* critical edge[1] *for f is an edge $g = (x, y) \in S(e)$ maximizing $\phi(f, g) := \dfrac{d_T(x, v) + w(f) + d_T(u, y)}{w(g)}$.*

DEFINITION 3 (Best Cut Edge). *A v-*best cut edge *for e (v-BCE) is an edge $f \in S(e, v)$ minimizing $\varphi_e(f) = \max_{g \in S(e)} \phi(f, g)$.*

Then, we will make use of the following property, which was given in [5]:

PROPOSITION 1. *Every v-BCE for e is a v-BSE for e.*

Let us first provide a high-level description of how we compute a v-BCE (i.e., a v-BSE) for e. The algorithm will compute $O(\log n)$ v-BCE *candidates*, the best of which will be a v-BCE for e. Informally speaking, each candidate f will be a swap edge close to the centroid of a certain subtree Λ of T. Depending on the position of a critical edge for f, the algorithm will recurse on a subtree of Λ and it will look for the next candidate. Thanks to the centroid decomposition of T, the number of recursions/candidates will then be $O(\log n)$ and, as we said, the best out of the $O(\log n)$ swap edge candidates will be a v-BSE for e.

[1] Notice that this definition does not contain $d_{G-e}(x, y)$ at the denominator, as expected, since it already incorporates the property stated in the forthcoming Proposition 1.

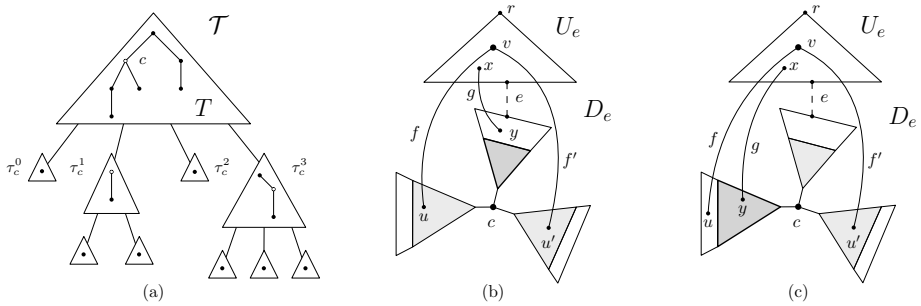

FIGURE 2. (a) An example of centroid decomposition of the tree T (which corresponds to the first vertex of $\mathcal{T}$). (b) and (c): Two of the four possible cases situation illustrated in Lemma 1. The subtree $\widehat{T}$ is represented by the three gray triangles along with the vertex c. f is a swap edge for e that minimizes $w(f) + d_T(u, c)$, and g is its corresponding critical edge. The (c, y)-*tree* of $\widehat{T}$ is drawn in bold. Notice that f and g do not need to be incident to $\widehat{T}$.

The key ingredient for the correctness of our algorithm is the next lemma. Given a subtree $\widehat{T}$ of T, a vertex $c \in V(\widehat{T})$, and a vertex $y \in V(T)$, consider the first vertex z of the unique path from y to c in T that also belongs to $V(\widehat{T})$. The $(c, y)-$*tree* of $\widehat{T}$ is defined as follows: (1) if $z = c$, then it is the empty tree; otherwise (2) it is the tree of the forest $\widehat{T} - c$ that contains z. Then, the following holds (see also Figure 2 (b) and (c)):

LEMMA 1. *Let $\widehat{T}$ be a subtree of T such that $V(\widehat{T}) \subseteq D_e$, and let $c \in V(\widehat{T})$. Moreover, let $f = (v, u) \in S(e, v)$ be a swap edge for e that minimizes $w(f) + d_T(u, c)$, and let $g = (x, y)$ be a critical edge for f. Assume that $S(e, v, V(\widehat{T}))$ contains a v-BCE for e. If f is not a v-BCE for e, then $S(e, v, V(T'))$ contains a v-BCE for e, where T' is the (c, y)-tree of $\widehat{T}$.*

PROOF. Suppose that f is not a v-BCE for e, we will show that no swap edge $f' = (v, u') \in S(e, v)$ with $u' \notin V(T')$ can be a v-BCE for e. Indeed:

$$\begin{aligned}
\varphi(f') \geq \phi(f', g) &= \frac{d_T(x, v) + w(f') + d_T(u', y)}{w(g)} \\
&= \frac{d_T(x, v) + w(f') + d_T(u', c) + d_T(c, y)}{w(g)} \\
&\geq \frac{d_T(x, v) + w(f) + d_T(u, c) + d_T(c, y)}{w(g)} \geq \phi(f, g) = \varphi(f),
\end{aligned}$$

Procedure FindBCE(Λ)

if $|V(\Lambda)| = 0$ **then return** $(\perp, \perp)$;
$c \leftarrow$ Centroid of Λ;
if $c \in U_e$ **then**
  $\tau \leftarrow$ unique child of Λ in $\mathcal{T}$ that contains all the vertices in $V(\Lambda) \cap D_e$;
  return FindBCE (τ);
else // $c \in D_e$
  Compute an edge $f = (v, u) \in \arg\min_{(v,u) \in S(e,v)} \{w(v, u) + d_T(u, c)\}$; // See Section 3.1.6
  $g_1 = (x, y) \leftarrow$ FindCritical(f, T); // Compute a critical edge for f (see Sec. 3.1.7)
  $\tau \leftarrow (c, y)$-tree of Λ; // Either τ is empty or it is a child of Λ in $\mathcal{T}$
  $(f', g_2) \leftarrow$ FindBCE (τ);
  if $\phi(f, g_1) \leq \phi(f', g_2)$ **then return** (f, g_1); **else return** (f', g_2);

where we used the fact that $d_T(u', y) = d_T(u', c) + d_T(c, y)$ as either $u' = c$ or u' and y are in two different connected components of $T - c$. □

Lemma 1 allows us to design a recursive algorithm for computing a v-BCE for e, whose key steps are highlighted in Procedure FindBCE(Λ). More precisely, the algorithm takes a tree Λ of the centroid decomposition $\mathcal{T}$ such that $V(\Lambda) \cap D_e \neq \emptyset$, and it computes a pair (f^*, g^*) such that if $S(e, v, V(\Lambda) \cap D_e)$ contains a v-BCE for e, then f^* is a v-BCE for e, and g^* is its critical edge. Algorithm FindBCE makes use of an additional function FindCritical(f, T) that returns a critical edge for f w.r.t. the failure of e. The initial call will be FindBCE(T). In order to handle base cases, we assume $\phi(\perp, \perp) = +\infty$.

We now prove the correctness of the procedure:

LEMMA 2. *Procedure FindBCE(T) computes a v-BCE for e.*

PROOF. Consider an invocation of the procedure and let Λ and (f^*, g^*) be its parameter and the edges it returns, respectively. We prove the following claim by induction on the cardinality of $V(\Lambda)$: if $S(e, v, V(\Lambda) \cap D_e)$ contains a v-BCE for e, then f^* is a v-BCE for e and g^* is a critical edge for f^*.

If $|V(\Lambda)| = 0$, then the claim trivially holds. Otherwise, $|V(\Lambda)| > 0$, and we distinguish two cases depending on the position of the centroid c of Λ. If $c \in U_e$, then there is only one child τ_c^j of Λ in $\mathcal{T}$ that contains all the vertices in

$V(\Lambda) \cap D_e$, as otherwise the vertices in D_e would be disconnected in Λ. Hence, if $S(e, v, V(\Lambda) \cap D_e)$ contains a v-BCE for e, then $S(e, v, V(\tau_c^j) \cap D_e)$ also contains a v-BCE for e, and the claim follows by the inductive hypothesis (as $|V(\tau_c^j)| < |V(\Lambda)|$). The remaining case is the one in which $c \in D_e$, here the claim follows from Lemma 1 (where now $\widehat{T}$ is the subtree of T induced by $V(\Lambda) \cap D_e$) together with the inductive hypothesis. □

The following lemma will provide an upper bound to the running time of the Procedure `FindBCE`:

LEMMA 3. *Procedure `FindBCE`(T) requires $O((\Gamma_f + \Gamma_{\mathtt{FC}}) \log n)$ time, where Γ_f and $\Gamma_{\mathtt{FC}}$ is the time required to perform Steps 7 and 8, i.e., the time to find edge f, and to execute procedure `FindCritical`, respectively.*

PROOF. First of all, notice that Step 4 can be performed in $O(1)$ time, after a $O(\log n)$ preprocessing time in which we mark all the nodes of $\mathcal{T}$ on the path between the leaf of $\mathcal{T}$ containing the lower vertex of e (which clearly belongs to D_e) and the root of $\mathcal{T}$. Then, we only need to bound the depth of the recursion of the call `FindBCE`(T). Observe that each time Procedure `FindBCE`(Λ) recursively invokes itself on a tree Λ', we have that Λ' is a child of Λ in $\mathcal{T}$. The claim follows since the height of $\mathcal{T}$ is $O(\log n)$. □

Actually, the time to execute Step 7 is $O(\log n)$, after a preprocessing time and space of $O(n^2)$, by making use of *top-trees* [3]. This result will be shown in the next subsection. On the other hand, Procedure `FindCritical` will require $O(\log^3 n)$ time and $O(m \log^2 n)$ space, as we will show in the next two subsections.

3.1.6. Selecting edge f in Step 7 of Procedure FindBCE(Λ).

In this section we show how to efficiently find an edge $f = (v, u) \in S(e)$ minimizing $w(f) + d_T(u, c)$, where c is a vertex in D_e, as Procedure `FindBCE` requires. We will show how this problem can be solved by using *top-trees* [3].

A top-tree is a dynamic data structure that maintains a (weighted) forest F of trees under *link* (i.e., edge-insertion) and *cut* (i.e., edge-deletion) operations. Moreover, some of the vertices of F can be *marked* and the top-tree is able to perform *closest marked vertex* (CMV, for short) queries, i.e., it can report the

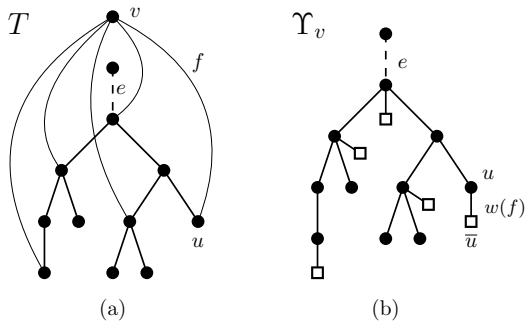

FIGURE 3. (a) The subtree of T induced by D_e along with swap edges in $S(e, v)$, and (b) the corresponding top-tree Υ_v. The black vertices of Υ_v are the same of the tree T. For each $u \in V(T)$ such that $f = (v, u) \in S(e, v)$, Υ_v contains an additional vertex $\overline{u}$ (shown as a white square), and a corresponding edge $(u, \overline{u})$ with weight $w(f)$.

marked vertex that is closest to a given vertex z. A top-tree on n vertices can be built in linear time and each of all the aforementioned operations requires $O(\log n)$ time.

We maintain a top-tree Υ_v of size $O(n)$ for each vertex $v \in V(T)$, and so we use a total of $O(n^2)$ space. Each of these top-trees is the tree T augmented with some additional marked vertices. More precisely, for each $v \in V(T)$ and for each edge $f = (v, u) \in E(G) \setminus E(T)$ we add to Υ_v a marked vertex $\overline{u}$ and the edge $(u, \overline{u})$ with a weight of $w(f)$ (see Figure 3).

Whenever we are finding a v-BSE for e and we need to find the edge f minimizing $w(f) + d_T(u, c)$ we do the following: (i) we cut the edge e from Υ_v, (ii) we perform a CMV query on Υ_v to find the closest marked vertex $\overline{u}$ to c, if any, (iii) we undo the cut operation by linking the endpoints of e in Υ_v, and finally (iv) we return the edge (v, u) (or $\perp$ if no $\overline{u}$ has been found).

LEMMA 4. *Let $e \in E(T)$ be a failing edge and let $c \in D_e$. An edge $f = (v, u) \in S(e, v)$ minimizing $w(f) + d_T(u, c)$ can be found in $O(\log n)$ time. Moreover, all the top-trees Υ_v can be initialized in $O(n^2)$ time.*

PROOF. Each of the n top-trees Υ_v can be built in time $O(n)$ by explicitly considering all the edges in $E(v)$ (notice that Υ_v contains at most $2n$ vertices as there can be at most one marked vertex per vertex in $V(T)$).

As for the time complexity of finding edge f, it immediately follows from the fact that we perform a constant number of link, cut, and CMV query operations, hence we only need to argue about correctness.

Notice that after we cut edge e from Υ_v in step (i), the tree T' of Υ_v containing c has exactly one (distinct) marked vertex $\overline{u}$ for each edge $(u, v) \in S(e, v)$. The claim follows as, by construction, the distance from c to $\overline{u}$ in T' is $d_{T'}(c, \overline{u}) = d_{T'}(c, u) + d_{T'}(u, \overline{u}) = d_T(c, u) + w(f)$. □

3.1.7. Computing a critical edge for f.

We will compute $O(\log^2 n)$ critical edge candidates for f and we will show that a critical edge for f will be one of them. More precisely, we look at $O(\log n)$ subtrees of the centroid decomposition $\mathcal{T}$ and, for each such a subtree Λ, we will consider $O(\log n)$ subtrees Ψ. Hence, we find a critical edge candidate having one endpoint in Ψ and the other in Λ. The choice of the $O(\log^2 n)$ pairs of trees will be guided by the position of f, while the computation of a candidate for a given pair (Ψ, Λ) is the core of the procedure and will be described in the next subsection.

DEFINITION 4 ((Ψ, Λ)-Critical Edge). *Given a failing edge e and a swap edge $f = (v, u) \in S(e, v)$, and given two trees Ψ, Λ of the centroid decomposition $\mathcal{T}$, a* (Ψ, Λ)-critical edge *for f is an edge g such that:*

$$g = (x, y) \in \arg \max_{g' \in S(e, V(\Psi) \cap U_e, V(\Lambda) \cap D_e)} \phi(f, g')$$

. When $\Psi = T$ we will refer to a (Ψ, Λ)-critical edge as a Λ-critical edge.

Let $f = (v, u) \in S(e, v)$ and let Λ be a tree of the centroid decomposition $\mathcal{T}$ such that $u \in V(\Lambda)$. Procedure `FindCritical` returns a Λ-critical edge for f, when edge e fails (such an edge always exists as f has one endpoint in U_e and the other in $V(\Lambda) \cap D_e$). Notice that the call `FindCritical`(f, T) in Procedure `FindBCE` computes a critical edge for f, since a T-critical edge for f is actually a critical edge for f.

Procedure `FindCritical` (see the next algorithm box) uses as a subroutine another procedure, named `FindCriticalCandidate`(f, Ψ, Λ), which for the sake of clarity will be exploited in the next subsection. For the moment, it

Procedure FindCritical($f = (v, u), \Lambda$)

1 **if** $V(\Lambda) = \{u\}$ **then return** `FindCriticalCandidate`(f, T, Λ) ;
2 $c \leftarrow$ Centroid of Λ;
3 Let j be the unique index in $\{0, 1, 2, 3\}$ such that $u \in V(\tau_c^j)$;
4 **if** $c \in U_e$ **then return** `FindCritical`(f, τ_c^j) ;
5 $\mathcal{G} \leftarrow \{$`FindCriticalCandidate`$(f, T, \tau_c^i) : i = 0, 1, 2, 3 \wedge i \neq j\}$; `// Here` $c \in D_e$
6 $g_1 \leftarrow \arg\max_{g \in \mathcal{G}} \phi(f, g)$;
7 $g_2 \leftarrow$ `FindCritical`(f, τ_c^j);
8 **return** $\arg\max_{g \in \{g_1, g_2\}} \{\phi(f, g)\}$;

is sufficient to know that `FindCriticalCandidate` receives three inputs, i.e., edge $f = (v, u)$ and two subtrees Ψ, Λ of the centroid decomposition $\mathcal{T}$ such that $v \in \Psi$ and, either $u \notin V(\Lambda)$ or Λ is the tree containing the sole vertex u, and it returns a (Ψ, Λ)-critical edge for f. If no such edge exists, then `FindCriticalCandidate` returns $\perp$ and we assume that $\phi(f, \perp) = -\infty$.

LEMMA 5. *Let $f = (v, u) \in S(e, v)$, and let Λ be a tree of the centroid decomposition $\mathcal{T}$ such that $u \in V(\Lambda)$. Procedure `FindCritical`(f, Λ) returns a Λ-critical edge for f.*

PROOF. The proof is by induction on the cardinality of $V(\Lambda)$.

If $|V(\Lambda)| = 1$, then the only vertex in Λ must be u, therefore the Procedure `FindCritical` invokes Procedure `FindCriticalCandidate`(f, T, Λ). Hence, assuming such a procedure is correct, it returns a (T, Λ)-critical edge, i.e., a Λ-critical edge. If $|V(\Lambda)| > 1$ then we distinguish two cases, depending on the position of the centroid c of Λ.

If $c \in D_e$ it is sufficient to notice that a Λ-critical edge for f must be incident to a tree τ_c^i for some $i = 0, 1, 2, 3$. Let j be the unique index in $\{0, 1, 2, 3\}$ such that $u \in V(\tau_c^j)$. If $j \neq i$ then, assuming Procedure `FindCriticalCandidate` is correct, it returns a (T, Λ)-critical edge g_1 (and hence a Λ-critical edge) for f. Procedure `FindCritical` then returns either g_1 or another edge g such that $\phi(f, g) = \phi(f, g_1)$. If $j = i$, the algorithm is recursively invoked and, since $|V(\tau_c^i)| < |V(\Lambda)|$ we know, by the induction hypothesis, that it correctly returns a τ_c^i-critical edge for f, which is also Λ-critical edge for f.

If $c \in U_e$, then we know that there is at most one τ_c^i that contains one or more vertices in D_e (as otherwise it would imply that the vertices in $V(\Lambda) \cap D_e$ are disconnected in Λ, a contradiction). Moreover, since $u \in V(\Lambda) \cap D_e$, there is exactly one such tree τ_c^i, namely τ_c^j. The algorithm recursively invokes itself on τ_c^j and, since $|V(\tau_c^j)| < |V(\Lambda)|$, we know, by the induction hypothesis, that it correctly returns a τ_c^j-critical edge for f, which is also Λ-critical edge for f. □

LEMMA 6. *Procedure* `FindCritical`(f, Λ) *requires* $O(\Gamma_{\texttt{FCC}} \cdot \log n)$ *time, where* $\Gamma_{\texttt{FCC}}$ *is the time required by an invocation of Procedure* `FindCriticalCandidate`*.*

PROOF. Notice that Procedure `FindCritical` performs exactly one recursive invocation for each vertex of the tree $\mathcal{T}$ on the unique path between the root of $\mathcal{T}$ and u in $\mathcal{T}$. The claim follows since the height of $\mathcal{T}$ is $O(\log n)$. □

In the next subsection, we show that $\Gamma_{\texttt{FCC}} = O(\log^2 n)$, and then we give our final result.

3.1.8. Procedure FindCriticalCandidate.

In this subsection, we describe the core of the procedure that computes a critical edge for f. Let us first describe informally the main idea of this part. Let $b \in U_e$ and $c \in D_e$, and consider any two edges $f = (v, u), g = (x, y) \in S(e)$ such that b (resp. c) is on the unique path from x to v (resp. from y to u) in T (see Figure 4). It turns out that the stretch factor of *any* f w.r.t. a *given* g can be though as a linear function $\Phi_{b,c,g}(t) = \alpha_{b,c}(g) \cdot t + \beta_{b,c}(g)$, where $\alpha_{b,c}(g)$ and $\beta_{b,c}(g)$ only depend on g. More precisely, we will have that $\phi(f, g) = \Phi_{b,c,g}(t_{b,c}(f))$, for a suitable value $t_{b,c}(f)$ which only depends on f. Hence, whenever we look for a critical edge for f, we can ask for a corresponding function $\Phi_{b,c,g}(t)$ with maximum value on $t_{b,c}(f)$. Since we do not know a priori the edge f for which we need to compute a critical edge, we will maintain this information as the *upper envelope* of a suitable set of functions. Let us make this idea more precise.

DEFINITION 5 (Upper Envelope). *Let* $\mathcal{F} = \{\Phi_1, \Phi_2, \ldots, \Phi_\ell\}$ *be a finite set of functions, where* $\Phi_i : \mathbb{R} \to \mathbb{R}$ *for every* $i = 1, 2, \ldots, \ell$*. The upper envelope of* $\mathcal{F}$ *is defined as* $\mathrm{UE}_{\mathcal{F}} : t \in \mathbb{R} \mapsto \arg\max_{\Phi \in \mathcal{F}} \Phi(t) \in 2^{\mathcal{F}}$.

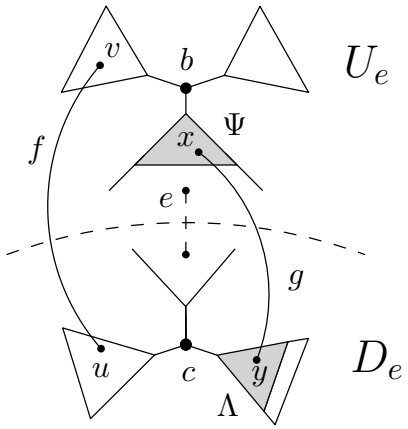

FIGURE 4. Illustration of Lemma 8. f is a swap edge for e, Ψ and Λ are two trees of the centroid decomposition, and b and c are their corresponding parent centroids. g is a potential (Ψ, Λ)-critical edge for f. Notice that the unique path from x to v (resp. from y to u) passes through b (resp. c).

Let $b \in U_e$ and $c \in D_e$. Given an edge $f = (v, u)$, define $t_{b,c}(f)$ as the quantity $d_T(b, v) + w(f) + d_T(u, c)$. Given an edge $g = (x, y)$, define $\alpha_{b,c}(g) = \frac{1}{w(g)}$ and $\beta_{b,c}(g) = \frac{d_T(x,b)+d_T(c,y)}{w(g)}$. Notice how, once b and c are fixed, $t_{b,c}(f)$ only depends on f while $\alpha_{b,c}(g)$ and $\beta_{b,c}(g)$ only depend on g. Let $\Phi_{b,c,g}(t) = \alpha_{b,c}(g) \cdot t + \beta_{b,c}(g)$.

LEMMA 7. *Let $f = (v, u) \in S(e, v)$. Let $b \in U_e$ and $c \in D_e$. Let X (resp. Y) be a set of vertices $x \in U_e$ (resp. $y \in D_e$) such that vertex b (resp. c) is on the unique path from x to v (resp. from y to u) in T. For every $g \in S(e, X, Y)$ we have $\phi(f, g) = \Phi_{b,c,g}(t_{b,c}(f))$.*

PROOF. Let $g = (x, y)$. We have:

$$\begin{aligned}
\phi(f, g) &= \frac{d_T(x, v) + w(f) + d_T(u, y)}{w(g)} \\
&= \frac{d_T(x, b) + d_T(b, v) + w(f) + d_T(u, c) + d_T(c, y)}{w(g)} \\
&= \frac{d_T(b, v) + w(f) + d_T(u, c)}{w(g)} + \frac{d_T(x, b) + d_T(c, y)}{w(g)} \\
&= \alpha_{b,c}(g) t_{b,c}(f) + \beta_{b,c}(g) = \Phi_{b,c,g}(t_{b,c}(f)).
\end{aligned}$$

□

DEFINITION 6 (Parent centroid). *Let τ be a tree of the centroid decomposition $\mathcal{T}$. The* parent centroid *of τ is the centroid of the parent of τ in $\mathcal{T}$.*

Lemma 7 is instrumental to proving the following (see Figure 4):

LEMMA 8. *Let $f = (v, u) \in S(e, v)$, and let Ψ, Λ be two trees of the centroid decomposition of T such that the following conditions hold: (i) $v \notin V(\Psi)$ or $V(\Psi) = \{v\}$, and (ii) $u \notin V(\Lambda)$ or $V(\Lambda) = \{u\}$. Let b (resp. c) be the parent centroid of Ψ (resp. Λ), and assume that $b \in U_e$ (resp. $c \in D_e$). Then, an edge g is a (Ψ, Λ)-critical edge for f if and only if $\Phi_{b,c,g} \in \mathrm{UE}_{\mathcal{F}}(t_{b,c}(f))$ where $\mathcal{F} = \{\Phi_{b,c,g'} : g' \in S(e, V(\Psi) \cap U_e, V(\Lambda) \cap D_e)\}$.*

PROOF. First of all we show the following property of the centroid decomposition $\mathcal{T}$: let $p, q \in V(T)$, and suppose that the unique path in $\mathcal{T}$ between the leaf nodes associated with p and q contains a node whose corresponding centroid is z. Then, the unique path between p and q in T contains z. Indeed, if z is either p or q, the property is trivially true. On the other hand, suppose that $z \notin \{p, q\}$, and let τ be the subtree of T associated with z in $\mathcal{T}$. Then, let τ_z^i be the child subtree of τ containing p. Observe that q is not in τ_z^i. Moreover, by construction, each path from a node of τ_z^i, and in particular from p, to any node outside τ_z^i, and in particular to q, must pass through z.

We now prove the claim. If $V(\Psi) = \{v\}$ (resp. $V(\Lambda) = \{u\}$) then the claim follows from Lemma 7 by choosing $X = \{v\}$ and $Y = V(\Lambda) \cap D_e$ (resp. $X = V(\Psi) \cap U_e$ and $Y = \{u\}$). The only remaining case to consider is the one in which $v \notin V(\Psi)$ and $u \notin V(\Lambda)$. Consider the vertices v and b (resp. u and c) in $\mathcal{T}$ and notice that v (resp. u) cannot be an ancestor of b (resp. c). Indeed, if v (resp. u) was an ancestor of b (resp. c) in $\mathcal{T}$, then subtree of T induced by the vertices in $V(\Psi)$ (resp. $V(\Lambda)$) would contain b (resp. c) contradicting the hypothesis. Hence, the path from any vertex in $V(\Psi)$ to v (resp. $V(\Lambda)$ to u) must traverse b (resp. c) in $\mathcal{T}$ and therefore the same holds in T. The claim follows by invoking Lemma 7 with $X = V(\Psi) \cap U_e$ and $Y = V(\Lambda) \cap D_e$. □

Lemma 8 allows us to design a recursive procedure to compute a (Ψ, Λ)-critical edge for f (see the next algorithm box). To this aim we will make use of a data structure $\mathcal{Q}_e$ that, for each edge $f \in S(e)$, and for each pair of

trees Ψ, Λ of the centroid decomposition, can perform a query operation that we name $\mathcal{Q}_e(f, \Psi, \Lambda)$. This query reports an edge whose function $\Phi_{b,c,g}$ is in $\mathrm{UE}_{\mathcal{F}}(t_{b,c}(f))$ where b and c are the parent centroids of Ψ and Λ, respectively, and $\mathcal{F} = \{\Phi_{b,c,g'} : g' \in S(e, V(\Psi) \cap U_e, V(\Lambda) \cap D_e)\}$.

Next two lemmas show the correctness and the running time of the procedure:

LEMMA 9. *Let be given an edge $f = (v, u) \in S(e, v)$ and two trees Ψ, Λ of the centroid decomposition such that: (i) $v \in V(\Psi)$, and (ii) $u \notin V(\Lambda)$ or $V(\Lambda) = \{u\}$. Then, it follow that Procedure `FindCriticalCandidate`(f, Ψ, Λ) computes a (Ψ, Λ)-critical edge for f.*

PROOF. First of all notice that if $V(\Lambda) \cap D_e = \emptyset$, then the algorithm correctly returns $\perp$.

We now prove the claim by induction on the cardinality of $V(\Psi)$. If $|V(\Psi)| = 1$, then the only vertex in Ψ must be v and Procedure `FindCriticalCandidate` queries $\mathcal{Q}_e$ for $\mathcal{Q}_e(f, \Psi, \Lambda)$. By Lemma 8, the returned edge is a (Ψ, Λ)-critical edge for f.

If $|V(\Psi)| > 1$ then we distinguish two cases, depending on the position of the centroid b of Ψ. If $b \in U_e$ it is sufficient to notice that a (Ψ, Λ)-critical edge for f must be incident to a tree τ_b^i for some $i = 0, 1, 2, 3$. Let j be the unique index in $\{0, 1, 2, 3\}$ such that $v \in V(\tau_b^j)$. If $j \neq i$ then, by Lemma 8, the query $\mathcal{Q}_e(f, \tau_b^i, \Lambda)$ returns a (τ_b^i, Λ)-critical edge g' (and hence g' is also a (Ψ, Λ)-critical edge) for f. Procedure `FindCritical` then returns either g or another edge g such that $\phi(f, g) = \phi(f, g')$. If $j = i$, the algorithm is recursively invoked and, since $|V(|\tau_b^i|)| < |V(\Psi)|$ we know, by the induction hypothesis, that it correctly returns a (τ_b^i, Λ)-critical edge for f, which is also (Ψ, Λ)-critical edge for f.

If $b \in D_e$, then we know that there is at most one τ_b^i containing one or more vertices in U_e (as otherwise it would imply that the vertices in $V(\Psi) \cap U_e$ are disconnected in Ψ, a contradiction). Moreover, since $v \in V(\Psi) \cap U_e$, there is exactly one such tree τ_b^i, namely τ_b^j. The algorithm recursively invokes itself on τ_b^j and, since $|V(\tau_b^j)| < |V(\Psi)|$, we know, by the induction hypothesis, that it correctly returns a τ_b^j-critical edge for f, which is also (Ψ, Λ)-critical edge for f. □

Procedure FindCriticalCandidate($f = (v, u), \Psi, \Lambda$)

if $V(\Lambda) \cap D_e = \emptyset$ **then return** $\perp$;
if $V(\Psi) = \{v\}$ **then return** $\mathcal{Q}_e(f, \Psi, \Lambda)$;
$b \leftarrow$ Centroid of Ψ;
Let j be the unique index in $\{0, 1, 2, 3\}$ such that $v \in V(\tau_b^j)$;
if $b \in D_e$ **then return** `FindCriticalCandidate`(f, τ_b^j, Λ);
$\mathcal{G} \leftarrow \{\mathcal{Q}_e(f, \tau_b^i, \Lambda) : i = 0, 1, 2, 3 \wedge i \neq j\}$; `// Here` $b \in U_e$
$g_1 \leftarrow \arg\max_{g \in \mathcal{G}} \phi(f, g)$;
$g_2 \leftarrow$ `FindCriticalCandidate`(f, τ_b^j, Λ);
return $\arg\max_{g \in \{g_1, g_2\}} \{\phi(f, g)\}$;

LEMMA 10. *Procedure* `FindCriticalCandidate`(f, Ψ, Λ) *requires a running time of* $O(\Gamma_{\mathcal{Q}_e} \cdot \log n)$*, where* $\Gamma_{\mathcal{Q}_e}$ *is the time required by a query on* $\mathcal{Q}_e$*.*

PROOF. Notice that Procedure `FindCriticalCandidate` performs exactly one recursive invocation for each vertex of the tree $\mathcal{T}$ on the unique path between the root of $\mathcal{T}$ and u in $\mathcal{T}$. The claim follows since the height of $\mathcal{T}$ is $O(\log n)$. □

Thus, to get the promised running time of $O(\log^2 n)$ for $\Gamma_{\texttt{FCC}}$, we are left to prove that $\Gamma_{\mathcal{Q}_e} = O(\log n)$. Actually, such a bound can be obtained by suitably implementing $\mathcal{Q}_e$ in such a way that all the underlying upper envelope functions are efficiently maintained, as follows.

3.1.9. Dynamic maintenance of the upper envelopes.

Procedure `FindCriticalCandidate` needs the auxiliary structure $\mathcal{Q}_e$. Explicitly building such a structure for each edge e would be too expensive. In the remaining of this section we show how all the $\mathcal{Q}_e$'s can be built in $O(m \log^4 n)$ time and $O(m \log^2 n)$ space. The idea is to exploit the order in which failing edges are considered, so as to reuse previously computed information to build $\mathcal{Q}_e$.

We implement $\mathcal{Q}_e$ as a dictionary of $O(n^2)$ elements that allows us to add, delete, and search for elements in $O(\log n)$ time per operation. Each element of $\mathcal{Q}_e$ is a data structure that can store a set $\mathcal{F}$ of linear functions and is able (i) to dynamically add/remove a function to/from $\mathcal{F}$ in $O(\log |\mathcal{F}|)$ time, (ii) given $t \in \mathbb{R}$, to report a function in $\mathrm{UE}_{\mathcal{F}}(t)$ in $O(\log |\mathcal{F}|)$ amortized time [14].

Each data structure in the dictionary is associated with a pair Ψ, Λ of trees of $\mathcal{T}$ and will contain all the functions $\Phi_{b,c,g}$ where b and c are the parent centroids of Ψ and Λ, respectively, and $g \in S(e, V(\Psi) \cap U_e, V(\Lambda) \cap D_e)$. We name such a structure $\mathcal{H}^e_{(\Psi,\Lambda)}$. The pair (Ψ, Λ) is also the key of $\mathcal{H}^e_{(\Psi,\Lambda)}$ in the dictionary.

Observe now that we can answer the query $\mathcal{Q}_e(f, \Psi, \Lambda)$ in $O(\log n)$ amortized time (the same query used in `FindCriticalCandidate`) , as follows: we search for $\mathcal{H}^e_{\Psi,\Lambda} \in \mathcal{Q}_e$ in $O(\log n)$ time, and then we perform a query operation on $\mathcal{H}^e_{\Psi,\Lambda}$ with $t = t_{b,c}(f)$ where b and c are the parent centroids of Ψ and Λ, respectively (see Lemma 7).

We now show how to build and maintain the $\mathcal{Q}_e$'s. Remember that we process the edges $e \in E(T)$ in a bottom-up fashion. Let T_e be the subtree of T induced by D_e. Whenever T_e consists of a single vertex, we build $\mathcal{Q}_e$ from scratch. If T_e contains 2 or more vertices then there are at most two edges e_1, $e_2 \in E(T_e)$ that are incident to e. We build $\mathcal{Q}_e$ by merging $\mathcal{Q}_{e_1}$ and $\mathcal{Q}_{e_2}$. This merge operation consists of a *join* step followed by an *update* step.

Whenever we add a function $\Phi_{b,c,g}$ to a structure $\mathcal{H}^e_{(\Psi,\Lambda)}$ of $\mathcal{Q}_e$ and we are either performing the update step or we are building $\mathcal{Q}_e$ from scratch, we say that we *insert* $\Phi_{b,c,g}$ into $\mathcal{Q}_e$. We associate a non-negative integer ν_e to $\mathcal{Q}_e$ that we call *virtual size* of $\mathcal{Q}_e$. The virtual size of $\mathcal{Q}_e$ is the overall number of inserts that have been performed either on $\mathcal{Q}_e$ itself or on any other $\mathcal{H}^{e'}_{(\Psi,\Lambda)}$ such that e' is an edge of T_e.

3.1.10. Building $\mathcal{Q}_e$ from scratch.

We start by creating an empty dictionary $\mathcal{Q}_e$ (initially $\nu_e = 0$). Since we are building $\mathcal{Q}_e$ from scratch, T_e contains only one vertex, say y. For each edge $g = (x, y) \in S(e, U_e, y)$, we explicitly consider all the pairs of trees (Ψ, Λ), such that Ψ contains x and Λ contains y, and we let b and c be the parent centroids of Ψ and Λ, respectively. We look for $\mathcal{H}^e_{(\Psi,\Lambda)}$ in the dictionary of $\mathcal{Q}_e$, if $\mathcal{H}^e_{(\Psi,\Lambda)}$ already exists, we add $\Phi_{b,c,g}$ to $\mathcal{H}^e_{(\Psi,\Lambda)}$. If $\mathcal{H}^e_{(\Psi,\Lambda)}$ is not found, we create a new empty structure $\mathcal{H}^e_{(\Psi,\Lambda)}$, we add $\Phi_{b,c,g}$ into $\mathcal{H}^e_{(\Psi,\Lambda)}$, and we add $\mathcal{H}^e_{(\Psi,\Lambda)}$ to $\mathcal{Q}_e$. In both cases we have that $\Phi_{b,c,g}$ is *inserted* into $\mathcal{Q}_e$ and hence we increase ν_e by 1.

3.1.11. Building $\mathcal{Q}_e$ by merging.

Let $e = (p, q)$ and remember that T_e contains more than 1 vertex. Since q has degree at most 3 in T, there are either 1 or 2 edges in T_e that are incident to q. Here we will discuss the case in which those edges are exactly 2 (as the case in which q is incident to only one edge is simpler).

Let e_1, e_2 be the two edges incident to q in T_e. We will merge $\mathcal{Q}_{e_1}$ and $\mathcal{Q}_{e_2}$ in order to obtain $\mathcal{Q}_e$. This operation is destructive, i.e., $\mathcal{Q}_{e_1}$ and $\mathcal{Q}_{e_2}$ will no longer exist at the end of the merge operation. Notice, however, that since we are processing the edges of T in a bottom-up fashion, $\mathcal{Q}_{e_1}$ and $\mathcal{Q}_{e_2}$ will no longer be needed by the algorithm.

We will now detail how the merge operation works. In fact, it consists of two steps: the *join step* and the *update step*.

3.1.11.1. *The join step.*

W.l.o.g., let $\nu_{e_1} \geq \nu_{e_2}$. We start by renaming $\mathcal{Q}_{e_1}$ to $\mathcal{Q}_e$ (so that all the structures that belong to the dictionary of $\mathcal{Q}_{e_1}$ that were named $\mathcal{H}^{e_1}_{(\Psi,\Lambda)}$ are now named $\mathcal{H}^{e}_{(\Psi,\Lambda)}$).

Now, for each structure $\mathcal{H}^{e_2}_{(\Psi,\Lambda)}$ in $\mathcal{Q}_{e_2}$, we first search for the structure $\mathcal{H}^{e}_{(\Psi,\Lambda)}$ in $\mathcal{Q}_e$ and, if such a structure is not found, we add new empty structure $\mathcal{H}^{e}_{(\Psi,\Lambda)}$ to $\mathcal{Q}_e$. Then, we *move* each function $\Phi_{b,c,g}$ in $\mathcal{H}^{e_2}_{(\Psi,\Lambda)}$ to $\mathcal{H}^{e}_{(\Psi,\Lambda)}$, i.e., we remove $\Phi_{b,c,g}$ from $\mathcal{H}^{e_2}_{(\Psi,\Lambda)}$ and, if $\Phi_{b,c,g}$ is not in $\mathcal{H}^{e}_{(\Psi,\Lambda)}$, we add it to $\mathcal{H}^{e}_{(\Psi,\Lambda)}$. Finally, after all the structures $\mathcal{H}^{e_2}_{(\Psi,\Lambda)}$ in $\mathcal{Q}_{e_2}$ have been considered, we destroy $\mathcal{Q}_{e_2}$ and we set ν_e to $\nu_{e_1} + \nu_{e_2}$.

3.1.11.2. *The update step.*

After the merge step is completed, $\mathcal{Q}_e$ contains all the functions corresponding to the edges g in $S(e_1) \cup S(e_2)$.

Notice, however, that all the edges (x, y) such that the *lowest common ancestor* (LCA) of x and y in T is q are both in $S(e_1)$ and $S(e_2)$ but they do not belong to $S(e)$, and hence they should not appear in $\mathcal{Q}_e$. On the converse, the edges in $S(e, U_e, q)$ are neither in $S(e_1)$ nor in $S(e_2)$ but they belong to $S(e)$, hence their corresponding functions should be added to $\mathcal{Q}_e$. This is exactly the goal of the update step.

We start by deleting the extra functions from $\mathcal{Q}_e$. We iterate over each edge $g = (x, y)$ such that the LCA of x and y is q and, for each pair of trees (Ψ, Λ) such that Ψ contains x and Λ contains y, we delete $\Phi_{b,c,g}$ from $\mathcal{H}^e_{(\Psi,\Lambda)}$ where b and c are the parent centroids of Ψ and Λ respectively. If $\mathcal{H}^e_{(\Psi,\Lambda)}$ becomes empty, we also delete $\mathcal{H}^e_{(\Psi,\Lambda)}$ from $\mathcal{Q}_e$.

We now add the missing functions to $\mathcal{Q}_e$. For each $g = (x, q) \in S(e, U_e, q)$, and for each pair of trees (Ψ, Λ), such that Ψ contains x and Λ contains q, we first search for $\mathcal{H}^e_{(\Psi,\Lambda)}$ in $\mathcal{Q}_e$ and, if it does not exist, we add new empty structure $\mathcal{H}^e_{(\Psi,\Lambda)}$ to $\mathcal{Q}_e$. Then, we add $\Phi_{b,c,g}$ to $\mathcal{H}^e_{(\Psi,\Lambda)}$, where b and c are the parent centroids of Ψ and Λ respectively. We increase ν_e by 1 to account for this insertion.

3.1.12. Analysis.

Here we bound the time required to dynamically maintain all the upper envelope structures.

LEMMA 11. *The overall number of distinct functions $\Phi_{b,c,g}$ ever inserted into at least one of the structures $\mathcal{Q}_e$ is $O(m \log^2 n)$.*

PROOF. Let us consider any edge $g = (x, y) \in E(G) \setminus E(T)$. If a function $\Phi_{b,c,g}$ associated with g is inserted into any $\mathcal{Q}_e$, this means that it added to some $\mathcal{H}^e_{(\Psi,\Lambda)}$ such that $x \in V(\Psi)$, $y \in V(\Lambda)$, and b (resp. c) is the parent centroids of Ψ (resp. Λ). Notice that there are $O(\log n)$ trees τ of the centroid decomposition $\mathcal{T}$ that contain x (resp. y), meaning that there are $O(\log^2 n)$ functions $\Phi_{b,c,g}$ associated to g. The claim follows by summing over all the edges in $E(G) \setminus E(T)$. □

LEMMA 12. *Each function $\Phi_{b,c,g}$ contributes at most 2 to the virtual size ν_e of any $\mathcal{Q}_e$.*

PROOF. It suffices to bound the overall number of insertions of $\Phi_{b,c,g}$ (regardless of the structure $\mathcal{Q}_e$ into which $\Phi_{b,c,g}$ is inserted). To this aim, consider the edge $g = (x, y)$ associated with $\Phi_{b,c,g}$ and, w.l.o.g., let x be the vertex that is closest to the root of T. Let also e_x (resp. e_y) be the edge from the parent of x to x (resp. from the parent of y to y) in T. We distinguish two cases depending

on the relative positions of x and y in T. If x is an ancestor of y, then $\Phi_{b,c,g}$ is only inserted in $\mathcal{Q}_{e_y}$. Indeed, g belongs to $S(e_y)$ but it does not belong to any $S(e')$ where $e' \in E(T_{e_y})$ is incident to e_y in T_{e_y}. For any other pair of edges $e'', e''' \in E(T)$ such that e''' is incident to e'' in $T_{e''}$ we have that either g does not belong to $S(e'')$, or it belongs to both $S(e'')$ and $S(e''')$, and hence $\Phi_{b,c,g}$ is not added to $\mathcal{Q}_{e''}$. If x is not an ancestor of y, then a similar argument shows that $\Phi_{b,c,g}$ can only be inserted in $\mathcal{Q}_{e_x}$ and in $\mathcal{Q}_{e_y}$. The claim follows. □

LEMMA 13. *Each function $\Phi_{b,c,g}$ is moved $O(\log n)$ times.*

PROOF. When a function is moved from any $\mathcal{Q}_{e_2}$ to $\mathcal{Q}_e$ it is because we are merging $\mathcal{Q}_{e_1}$ with $\mathcal{Q}_{e_2}$, where e_1 and e_2 are edges incident to e in T_e. Notice that, before the merge operation takes place, we must have $\nu_{e_1} \geq \nu_{e_2}$ and hence, at the end of the merge operation, $\nu_e \geq \nu_{e_1} + \nu_{e_2} \geq 2\nu_{e_2}$. In other words, each time a function $\Phi_{b,c,g}$ is moved, we have that the *virtual size* of the structure to which $\Phi_{b,c,g}$ belongs at least doubles. Therefore, after a function has been moved r times, the structure containing $\Phi_{b,c,g}$ must have a virtual size of at least 2^r.

Notice now that Lemma 11 and Lemma 12 both imply an upper bound of $O(m \log^2 n)$ to the virtual size of any $\mathcal{Q}_e$. We can conclude that a function can be moved $O(\log(m \log^2 n)) = O(\log n)$ times. □

PROPOSITION 2. *The total time spent building and merging all the data structures $\mathcal{Q}_e$ is $O(m \log^4 n)$.*

PROOF. From Lemma 11 and Lemma 12 we have that the total number of insertions of functions $\Phi_{b,c,g}$ into the structures $\mathcal{H}^e_{(\Psi,\Lambda)}$ is $O(m \log^2 n)$ and, since each insertion requires time $O(\log n)$, the total time spent due to insertions is $O(m \log^3 n)$. Moreover, since each function is deleted at most once, and a deletion takes $O(\log n)$ time, we have that the total time spent for deleting functions is $O(m \log^3 n)$.

Concerning moving of functions, by Lemma 13 we have that every function is moved $O(\log n)$ times. Since there are $O(m \log^2 n)$ functions, as shown by Lemma 11, and a function can be moved in $O(\log n)$ time, we have that the total time spent moving functions is $O(m \log^4 n)$. □

Summarizing, by wrapping up all the lemmas, and by observing that we need $O(n^2)$ space to handle the top-trees, and $O(m \log^2 n)$ space to implement each $\mathcal{Q}_e$, we eventually can give the following:

THEOREM 1. *The ABSE-TS problem can be solved in $O(n^2 \log^4 n)$ time and $O(n^2 + m \log^2 n)$ space.*

PROOF. First of all, notice that the analysis contained in Section 3.1.12, shows that $\Gamma_{\mathcal{Q}_e} = O(\log n)$ (in an amortized sense), and moreover that the time needed to build and to maintain $\mathcal{Q}_e$ (i.e., the underlying upper envelope functions) is $O(m \log^4 n)$. Thus, the overall time complexity follows from the fact that we solve $O(n^2)$ subproblems (i.e., we find a v-BSE for every tree edge e), and each one of them costs $O(\log^4 n)$ time from Lemmas 3, 6, 10, and from $\Gamma_{\mathcal{Q}_e} = O(\log n)$. In turn, each element of such a dictionary is a data structure storing a set $\mathcal{F}$ of linear functions and is able to (i) dynamically add/remove a function to/from $\mathcal{F}$ in $O(\log |\mathcal{F}|)$ time, (ii) given $t \in \mathbb{R}$, report a function in $\mathrm{UE}_{\mathcal{F}}(t)$ in $O(\log |\mathcal{F}|)$ time [14]. Altogether, the space occupancy will be $O(n^2 + m \log^2 n)$, as a result of $O(n^2)$ space to handle the top-trees, and of $O(m \log^2 n)$ space needed to implement each $\mathcal{Q}_e$. □

3.2. Unweighted case

In this section, we provide an algorithm for unweighted graphs running in $O(mn \log n)$ time and using $O(m)$ space. A high-level description of the algorithm is the following: For each vertex $z \in V$, it first computes a *candidate* best swap edge for each edge e of T among the non-tree edges incident to z, if any. As we will see, this step can be performed in $O(m \log n)$ time and $O(m)$ space, and it will be repeated n times (once for each vertex $z \in V$). Then, a best swap edge of e is computed by selecting, among the candidate best swap edges of e, a non-tree edge whose corresponding swap tree is of minimum stretch. To optimize space consumption, the algorithm does not explicitly store the set of all the candidate best swap edges but, once a (new) candidate best swap edge of a tree edge e is computed, the algorithm only updates the best swap edge of e found so far in constant time.

Let us now give a detailed description of the algorithm. We fix a vertex $z \in V$ and a tree edge $e = (u, v)$, and we show that the problem of computing the candidate best swap edge of e among the non-tree edges incident to z reduces to a problem instance of the *subset minimum eccentricity problem on trees*. In the *subset minimum eccentricity problem on trees*, we are given a tree T', with a positive real cost $w'(e')$ associated with each edge e' of the tree, and a set $U \subseteq V(T')$, and we are asked to find a vertex of U of minimum *eccentricity*.[2]

The reduction works as follows. Let T_u and T_v be the two subtrees obtained by removing e from T and containing u and v, respectively. Let $\widetilde{V}$ be the subset of vertices of T_v which are incident to some non-tree edge in $C(e)$, let $\widehat{T}$ be the subtree of T_v rooted at v and induced by all the paths from v towards all the vertices of $\widetilde{V}$. If $\widehat{T}$ is not a path, then let r_v be the closest child of v having two or more children in $\widehat{T}$ (possibly, r_v=v), otherwise let r_v be equal to (unique) leaf of $\widehat{T}$. Finally, let $\widetilde{T}$ be the subtree of $\widehat{T}$ rooted at r_v, let $E(y)$ the set of non-tree edges in $C(e)$ incident to y, and let

$$\omega_y = \max_{(x,y)\in E(y)} \left\{ d_T(z, x) + 1 \right\}$$

be the maximum distance from z to y w.r.t. all the trees obtained by swapping e with a non-tree edge incident to y. The instance of the subset tree center problem is defined as follows. The tree T' is an unrouted copy of $\widetilde{T}$ augmented by the addition of a new vertex l_y and the edge (y, l_y) for each vertex $y \in \widetilde{V}$; $U = \{y \mid (z, y) \in E(z)\}$ is the set of endvertices in T_v of non-tree edges incident to z (observe that vertex z is not taken into account). Finally, the cost function of a tree edge e' of T' is

$$w'(e') := \begin{cases} \omega_y & \text{if } e' = (y, l_y); \\ 1 & \text{if } e' \in E(\widetilde{T}). \end{cases}$$

Observe that set of the leaves of T' is $\{l_y \mid y \in \widetilde{V}\}$. Next lemma shows the link between the problem of finding a candidate best swap edge of e and the problem of finding the vertex of U having minimum eccentricity in T'.

[2]The *eccentricity* of a vertex a of T' is equal to $\max_{b\in V(T')} d_{T'}(a, b)$.

LEMMA 14. *Let $f = (u', v') \in E(z)$, with $u' = z$. The value of formula (See 2) computed w.r.t. f is equal to the eccentricity of v' in T'.*

PROOF. Indeed, since $u' = z$, (See 2) can be rewritten as

$$\begin{aligned}
&\max_{g=(x,y)\in C(e)} \left\{ \frac{d_T(x,z) + w(f) + d_T(y,v')}{w(g)} \right\} \\
&= \max_{(x,y)\in C(e)} \left\{ d_T(x,z) + 1 + d_T(y,v') \right\} \\
&= \max_{y\in \widetilde{V}} \left\{ \max_{(x,y)\in E(y)} \left\{ d_T(x,z) + 1 + d_T(y,v') \right\} \right\} \\
&= \max_{y\in \widetilde{V}} \left\{ d_T(v',y) + \max_{(x,y)\in E(y)} \left\{ d_T(x,z) + 1 \right\} \right\} \\
&= \max_{y\in \widetilde{V}} \left\{ d_{T'}(v',y) + \omega_y \right\} \\
&= \max_{y\in \widetilde{V}} \{ d_{T'}(v', l_y) \} = \max_{b\in V(T')} \{ d_{T'}(v', b) \},
\end{aligned}$$

where the first equality holds because G is unweighted, the second equality holds because $C(e) = \bigcup_{y\in\widetilde{V}} E(y)$, while the last equality holds because the eccentricity of any vertex of T' is given by the length of a path towards some leaf of T', i.e., some l_y. □

The subset minimum eccentricity problem on trees is linear time solvable via a dynamic programming algorithm that computes the vertex eccentricities. Therefore, Lemma 14 already implies an $O(n^2)$ time and $O(m)$ space algorithm for the problem of computing a candidate best swap edge of every tree edge, as this problem is equivalent to solving $n-1$ instances of the subset minimum eccentricity problem on trees (one for each tree edge failure). However, for each vertex z, we can reduce the time complexity of computing a candidate best swap edge of every tree edge to $O(m \log n)$ by exploiting the similarities between instances of the subset minimum eccentricity problem on trees induced by the failure of adjacent tree edges. More precisely, for each vertex z, the algorithm roots T at z, visits the tree edges in preorder, and uses a *top tree* to efficiently generate and solve the corresponding $n-1$ instances of the subset minimum eccentricity problem on trees.

A *top tree* (see [3]) is a data structure that maintains a dynamic forest on a fixed set of N vertices, some of which are marked, can be initialized as an empty forest in $O(N)$ time and space, and supports each of the following operations in $O(\log N)$ time:

`cut`$(\bar{e})$**::** if $\bar{e}$ is an edge of the forest, it removes the edge $\bar{e}$ from the forest;

`link`(a, b, ω)**::** if a and b are vertices of different trees in the forest, it adds the edge (a, b) of cost ω to the forest;

`increase-cost`(a, b, ω)**::** if (a, b) is an edge of the forest of cost ω', it updates the cost of (a, b) to $\max\{\omega', \omega\}$; otherwise this operation is equivalent to `link`(a, b, ω);[3]

`center`(a)**::** it returns a triple $(c, \bar{a}, \bar{b})$, where c is a *center* of the tree,[4] say T'', in the forest that contains a, while $\bar{a}$ and $\bar{b}$ are the two endpoints of a *diametral path* of T'';[5]

`node`(a, b, k)**::** if a and b are vertices of the same tree, say T'', in the forest and k is a positive integer upper bounded by the hop-distance from a to b in T'', it returns the k-st node along the path from a to b in T'';

`closest`(a)**::** it returns a marked vertex which is closest to a (w.r.t. forest distances);

`distance`(a, b)**::** it returns the distance in the forest from a to b.

The algorithm uses the top tree as follows (see Algorithm 1 for the details and Figure 5 for an example). At the beginning of the visit of z the top tree is initialized as an empty forest of $2n$ vertices, where there are two vertices y and l_y for each $y \in V$. Furthermore, the set of marked vertices is $\{y \mid (z, y) \in E(z)\}$.

Let $e = (u, v)$ be the failing tree edge and, w.l.o.g., assume that z is a node of T_u. If e is not incident to z, i.e., $z \neq u$, then let e' be the tree edge incident to u along the path from z to u in T. Finally, let $\widetilde{C}(e) = C(e)$ if $u = z$, and $\widetilde{C}(e) = C(e) \setminus C(e')$ otherwise.

For every $(x, y) \in \widetilde{C}(e)$, with $x \in V(T_u)$ and $y \in V(T_v)$, the algorithm first increases the cost of the edge (y, l_y) to $d_T(z, x) + 1$ and then updates the top

[3]The `increase-cost` operation is not a basic primitive of the top tree, but it can be easily implemented via a cut operation (that has to be modified to return the cost of the removed edge, if any) followed by a link operation.

[4]A tree *center* is a vertex of the tree of minimum eccentricity.

[5]A *diametral path* is a path of the tree of maximum length.

tree by adding all the missing edges of the path from y to r_v in T, where the cost of each missing edge is 1. Next, the algorithm removes e from the top tree. Finally, it computes a candidate best swap edge of e using a suitable combination of `center`, `node`, `closest`, and `distance` operations according to the following four lemmas which show the relationships among tree center(s), diametral path(s), and vertex eccentricities, as well as an interesting connection between solutions of instances of the subset minimum eccentricity problem on trees generated by the algorithm and tree center(s).

LEMMA 15 (folklore). *Any positively edge-weighted tree has either one center or two centers. Furthermore, if the tree has two centers, say c and c', then (c, c') is an edge of the tree.*

LEMMA 16 (folklore). *Any diametral path of a positively edge-weighted tree contains all the tree centers.*

The following lemma is a stronger version of Lemma 16.

LEMMA 17 ([11]). *Let a be a vertex of a positively edge-weighted tree T'' and let $\bar{a}$ and $\bar{b}$ be the endvertices of a diametral path of T''. The eccentricity of a is equal to $\max\{d_{T''}(a, \bar{a}), d_{T''}(a, \bar{b})\}$. Furthermore, the longest path in T'' between the one from a to $\bar{a}$ and the one from a to $\bar{b}$ contains all the tree centers.*

LEMMA 18. *For a fixed vertex $z \in V$ and a fixed tree edge $e \in E(T)$, an optimal solution of the corresponding instance $\langle T', w', U\rangle$ of the subset minimum eccentricity problem on trees is a vertex of U that minimizes the distance from its closest tree center.*

PROOF. By construction, it is easy to see that the instance $\langle T', w', U\rangle$ satisfies the following properties:

(i) edge costs are positive integers;

(ii) all edges of cost strictly greater than 1 are incident to leaves of T', i.e., the vertices l_y's.

We prove the claim by cases according to the number of centers of T'. Thus, according to Lemma 15, we have to distinguish between the following two cases: T' has two centers and T' has exactly one center.

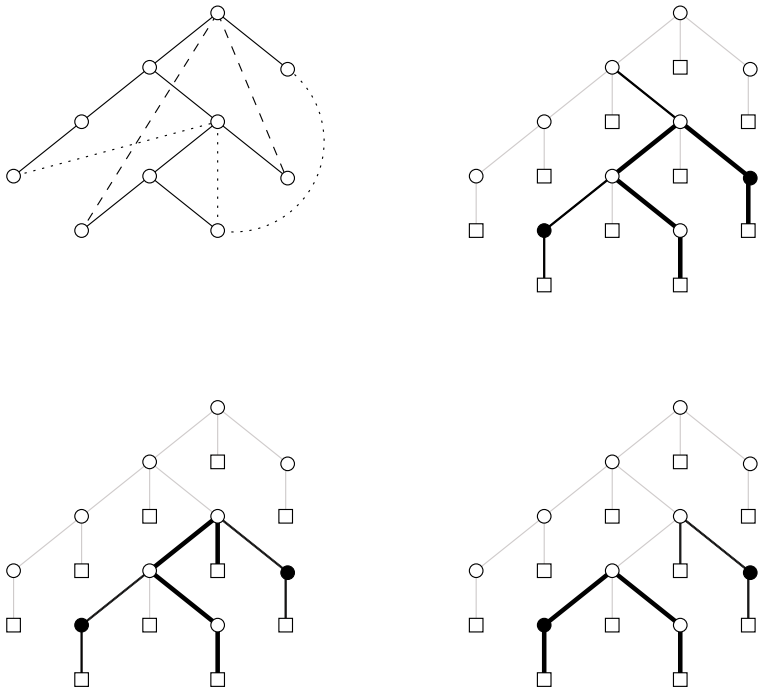

FIGURE 5. An illustration of the execution of Algorithm 1 during the visit of vertex z. The input instance of the ABSE spanner tree problem is shown in (a). The solid edges are the tree edges, the dashed edges are the non-tree edges incident to z, and the dotted edges are the other non-tree edges. The top trees corresponding to the failure of the tree edges e'', e', and e are shown in (b), (c), and (d), respectively. The square vertices correspond to the l_y's vertices. The marked vertices are of black color while the white vertices are unmarked. The solid edges denote the underlying top tree structure, the black edges are the edges of the top tree. For each failing edge, the corresponding tree in the top tree shows the diametral path (in bold), the tree center (vertex c) and the marked vertex which minimizes the distance from its closest tree center (vertex s^*). The candidate best swap edge of e'' is f which induces a tree having stretch factor equal to 4. The candidate best swap edge of e' is f' which induces a tree having stretch factor equal to 5. Finally, the candidate best swap edge of e is f which induces a tree having stretch factor equal to 5.

We begin with the case in which T' has two centers, say c and c'. Let $\bar{a}$ and $\bar{b}$ be the endvertices of a diametral path of T'. By Lemma 16, the path from $\bar{a}$ to $\bar{b}$ contains both c and c'. W.l.o.g., we assume that $d_{T'}(\bar{a}, c) < d_{T'}(\bar{a}, c')$. By Lemma

17, the eccentricities of c and c' are equal to $\max\left\{d_{T'}(c,\bar{a}), d_{T''}(c,\bar{b})\right\} = d_{T'}(\bar{b},c)$ and $\max\left\{d_{T'}(c',\bar{a}), d_{T''}(c',\bar{b})\right\} = d_{T'}(\bar{a},c')$, respectively. Furthermore, as c and c' are both tree centers, we have that $d_{T'}(c,\bar{a}) = d_{T'}(c',\bar{b})$. By Lemma 17, the eccentricity of a vertex $a \in U$ is therefore equal to

$$\begin{aligned}\max\left\{d_{T'}(a,\bar{a}), d_{T'}(a,\bar{b})\right\} &= \min\left\{d_{T'}(a,c) + d_{T'}(c,\bar{b}), d_{T'}(a,c') + d_{T'}(c',\bar{a})\right\} \\ &= d_{T'}(c,\bar{a}) + \min\left\{d_{T'}(a,c), d_{T'}(a,c')\right\},\end{aligned}$$

where the first equality can be easily proved by case analysis (it clearly holds in the case $d_{T'}(a,c) \leq d_{T'}(a,c')$ as well as in the complementary case $d_{T'}(a,c) > d_{T'}(a,c')$). Since $d_{T'}(c,\bar{a})$ does not depend on a, the above equality implies that a vertex of U having minimum eccentricity is then equal to a vertex of U that minimizes the distance from its closest tree center. Hence, we have proved the claim for the case in which T' has two centers.

Consider now the case in which T' has only one center, say c. Let $\bar{a}$ and $\bar{b}$ be the endvertices of a diametral path of T'. By Lemma 16, the path from $\bar{a}$ to $\bar{b}$ contains c. W.l.o.g., we assume that $d_{T'}(c,\bar{a}) \geq d_{T'}(c,\bar{b})$. We divide the proof into the following two cases: $d_{T'}(c,\bar{a}) = d_{T'}(c,\bar{b})$ and $d_{T'}(c,\bar{a}) > d_{T'}(c,\bar{b})$.

In the former case, i.e., $d_{T'}(c,\bar{a}) = d_{T'}(c,\bar{b})$, by Lemma 17, the eccentricity of a vertex $a \in U$ is upper bounded by

$$\begin{aligned}&\max\left\{d_{T'}(a,\bar{a}), d_{T'}(a,\bar{b})\right\} \\ &\leq \max\left\{d_{T'}(a,c) + d_{T'}(c,\bar{a}), d_{T'}(a,c) + d_{T'}(c,\bar{b})\right\} \\ &= d_{T'}(a,c) + d_{T'}(c,\bar{a})\end{aligned}$$

and, since the path in T' from c to $\bar{a}$ is edge-disjoint w.r.t. the path in T' from c to $\bar{b}$ are edge-disjoint, it is lower bounded by

$$\begin{aligned}&\max\left\{d_{T'}(a,\bar{a}), d_{T'}(a,\bar{b})\right\} \\ &\geq \min\left\{d_{T'}(a,c) + d_{T'}(c,\bar{a}), d_{T'}(a,c) + d_{T'}(c,\bar{b})\right\} \\ &= d_{T'}(a,c) + d_{T'}(c,\bar{a}).\end{aligned}$$

Therefore, $\max\left\{d_{T'}(a,\bar{a}), d_{T'}(a,\bar{b})\right\} = d_{T'}(a,c) + d_{T'}(c,\bar{a})$. Since $d_{T'}(c,\bar{a})$ does not depend on a, the above equality implies that a vertex of U having minimum eccentricity is then equal to a vertex of U that minimizes the distance from c.

In the latter case, i.e. when $d_{T'}(c,\bar{a}) > d_{T'}(c,\bar{b})$, by property (i) we have that $d_{T'}(c,\bar{a}) \geq d_{T'}(c,\bar{b}) + 1$. As a consequence, and using property (ii), we have that $(c,\bar{a})$ is an edge of T', otherwise the eccentricity value of the vertex immediately following c along the path from c to $\bar{a}$ would be strictly smaller than the eccentricity of c. Therefore, by Lemma 17, the eccentricity of a vertex $a \in U$ is upper bounded by

$$\begin{aligned}
&\max\left\{d_{T'}(a,\bar{a}), d_{T'}(a,\bar{b})\right\} \\
&\leq \max\left\{d_{T'}(a,c) + d_{T'}(c,\bar{a}), d_{T'}(a,c) + d_{T'}(c,\bar{b})\right\} \\
&= d_{T'}(a,c) + d_{T'}(c,\bar{a}).
\end{aligned}$$

Furthermore, since $a \in U$, and vertices of U are internal vertices of T', we have that $a \neq \bar{a}$. As a consequence, the path in T' from a to c does not contain the edge $(\bar{a},c)$. Therefore, by Lemma 17, the eccentricity of a is lower bounded by

$$\max\left\{d_{T'}(a,\bar{a}), d_{T'}(a,\bar{b})\right\} \geq d_{T'}(a,c) + +d_{T'}(c,\bar{a}).$$

Thus, $\max\left\{d_{T'}(a,\bar{a}), d_{T'}(a,\bar{b})\right\} = d_{T'}(a,c) + d_{T'}(c,\bar{a})$. Once again, as $d_{T'}(c,\bar{a})$ is independent of a, the above equality implies that a vertex of U having minimum eccentricity corresponds to a vertex of U that minimizes the distance from c. This concludes the proof. □

We can now prove the main theorem.

THEOREM 2. *Algorithm 1 solves the ABSE tree spanner problem on unweighted graphs in $O(mn\log n)$ time and $O(m)$ space.*

PROOF. For every vertex $z \in V$ and for every tree edge $e \in E(T)$, Algorithm 1 reduces the problem of computing the candidate best swap edge of e, among the non-tree edges incident to z, to an instance of the subset minimum eccentricity problem on trees and then solves the latter instance so as explained in Lemma 18. Therefore, the algorithm correctness follows by the aforementioned lemma and by Lemma 14. Concerning the time and space complexity of

the algorithm, first of all observe that the space consumption is clearly linear in m. To prove the $O(mn \log n)$ time bound, we show that for a fixed vertex $z \in V$, the algorithm runs in $O(m \log n)$ time.

Let $z \in V$ be fixed. Observe that all the sets $\widetilde{C}(e)$ can be computed in $O(m)$ time by computing the *least common ancestors* for the endpoints of each non-tree edge (x, y) in constant time (see [37]) and determining the at most two edges along the path from x to y in T which are incident to the least common ancestor of x and y. This implies that each non-tree edge (x, y) is contained in at most two sets among all $\widetilde{C}(e)$'s. The top tree of $2n$ vertices can be initialized in $O(n)$ time (see [3]).

To prove the claim, it remains to show that the overall number of operations performed on the top tree is at most $O(m)$. First of all, observe that the number of `cut`, `center`, `closest`, `distance`, and `node` operations is constant for each tree edge $e \in E(T)$. Therefore, the overall number of `cut`, `center`, `closest`, `distance`, and `node` operations, for any vertex $z \in V$, is $O(n)$. As there is exactly an `increase-cost` operation for each non-tree edge $g \in \widetilde{C}(e)$ and since $\sum_{e \in E(T)} |\widetilde{C}(e)| \leq 2m$, the overall number of `increase-cost` operations, for any $z \in V$, is $O(m)$. Finally, as the top tree contains $2n$ vertices, since the overall number of `cut` operations is $2(n-1)$ (at most two `cut` operations for each tree edge failure), and since a `link` operation is performed only when an edge is missing, the overall number of `link` operations, for any vertex $z \in V$, is at most $4n-3$ as it can never exceed $2n-1$ (the size of a tree spanning $2n$ vertices) plus the at most $2(n-1)$ `cut` operations. □

Algorithm 1: Algorithm for ABSE spanner tree on unweighted graphs.

Input : a 2-edge-connected graph $G = (V, E)$ and a spanning tree T of G.
Output: for every $e \in E(T)$, a best swap edge that minimizes the stretch factor of the swap tree.

for *every* $e \in E(T)$ **do**
  $f_e = \perp$; /* f_e **stores the best swap edge of** e */
  $\mu_e = +\infty$; /* μ_e **stores the value of formula (See 2) w.r.t.** f_e */
for *every* $z \in V$ **do**
  root T at z;
  compute the set $\widetilde{C}(e)$ for every $e \in E(T)$;
  initialize the top tree as an empty forest on a set $\mathcal{V}$ of $2n$ vertices, where there are two vertices y and l_y for each vertex $y \in V$, and mark all the vertices of the set $\{y \in \mathcal{V} \mid (z, y) \in E \setminus E(T)\}$;
  /* **We assume that every tree in the top tree is rooted and that the parent of a root vertex is the root vertex itself. The boolean vector** B **keeps track of the vertices of** T **that are connected to their parents in the underlying structure of the top tree (see Figure 5) so as to have a frugal usage of** `link` **operations. The vector element** $r[a]$ **stores the closest child of** a **of degree** ≥ 2 **in the tree of the top tree that contains** a**, say** T''**. If such a vertex does not exist, then** $r[a]$ **stores the parent of the (unique) leaf of** T''**.** */
  for *every* $a \in V$ **do** $B[a] = 0$; $r[a] = a$;
  for *every* $e = (u, v) \in E(T)$ *in preorder w.r.t.* z **do**
    /* **instance of the subset min eccentricity problem on trees is built** */
    for *every* $(x, y) \in \widetilde{C}(e)$ **do**
      w.l.o.g., let $y \in V(T_v)$; $\texttt{increase-cost}\big(y, l_y, d_T(z, x) + 1\big)$;
      while $B[y] = 0$ *and* $y \neq v$ **do**
        let p be the parent of y in T; $\texttt{link}\big(y, p, 1\big)$; $B[y] = 1$;
        if p has exactly one child **then** $r[p] = r[y]$;
        if p has exactly two children **then** $r[p] = p$;
        $y = p$;
    $\texttt{cut}(e)$;
    if u has no child **then** $r[u] = u$;
    if u has exactly one child **then** $r[u] = y$, where y is the (unique) child of u;
    /* **updating the vector** r **after the execution of the following instruction is unnecessary as, for each** $\texttt{cut}((r[v], p))$**, the corresponding** $\texttt{link}(r[v], p, 1)$ **is executed at the end of the for loop** */
    if $r[v] \neq v$ **then** $\texttt{cut}((r[v], p))$, where p is the parent of $r[v]$ in T;
    /* **instance of the subset min eccentricity problem on trees is solved** */
    $(c, \bar{a}, \bar{b}) = \texttt{center}(r[v])$;
    /* **a marked vertex** s **that minimizes the distance from center** c**, and the corresponding eccentricity value are computed** */
    $s = \texttt{closest}(c)$; $\mu = \max\Big\{\texttt{distance}(s, \bar{a}), \texttt{distance}(s, \bar{b})\Big\}$;
    /* **the second (potential) tree center** c' **is computed.**/
    if $\texttt{distance}(c, \bar{a}) \leq \texttt{distance}(c, \bar{b})$ **then** $c' = \texttt{node}(c, \bar{b}, 1)$; **else** $c' = \texttt{node}(c, \bar{a}, 1)$;
    /* **a marked vertex** s' **that minimizes the distance from vertex** c'**, and the corresponding eccentricity value are computed** */
    $s' = \texttt{closest}(c')$; $\mu' = \max\Big\{\texttt{distance}(s', \bar{a}), \texttt{distance}(s', \bar{b})\Big\}$;
    /* **the marked vertex** s^* **of minimum eccentricity, and its corresponding eccentricity value are computed.** */
    if $\mu \leq \mu'$ **then** $s^* = s$; $\mu^* = \mu$; **else** $s^* = s'$; $\mu^* = \mu'$;
    /* **the best swap edge of** e **is updated** */
    if $\mu^* < \mu_e$ **then** $\mu_e = \mu^*$; $f_e = (z, s^*)$;
    if $r[v] \neq v$ **then** $\texttt{link}(r[v], p, 1)$, where p is the parent of $r[v]$ in T;
return $\{f_e \mid e \in E(T)\}$;

CHAPTER 4

Swapping on a Shortest Path Tree

4.1. Preliminary definitions

Let $G = (V(G), E(G), w)$ be a 2-edge-connected, edge-weighted, and undirected graph with cost function $w : E(G) \to \mathbb{R}^+$. We denote by n and m the number of vertices and edges of G, respectively.

Let T be an SPT of G rooted at $s \in V(G)$. Given an edge $e \in E(T)$, we let $C(e)$ be the set of all the *swap edges* for e, i.e., all edges in $E(G) \setminus \{e\}$ whose endpoints lie in two different connected components of $T - e$, and let $C(e, X)$ be the set of all the swap edge for e incident to a vertex in $X \subseteq V(G)$. For any $e \in E(T)$ and $f \in C(e)$, let $T_{e/f}$ denote the *swap tree* obtained from T by replacing e with f. Let $T_v = (V(T_v), E(T_v))$ be the subtree of T rooted at $v \in V(G)$. Give a swap edge $f = (x, y)$, we assume that the first appearing endvertex is the one closest to the source, and we may denote by $w(x, y)$ its weight. We define the *stretch factor of* y *w.r.t.* s, T, G as $\sigma_G(T, y) = \frac{d_T(s,y)}{d_G(s,y)}$.

Given an SPT T of G, the `ABSE-MS` problem is that of finding, for each edge $e = (a, b) \in E(T)$, a swap edge f^* such that:

$$f^* \in \arg\min_{f \in C(e)} \left\{ \mu(f) := \max_{v \in V(T_b)} \sigma_{G-e}\big(T_{e/f}, v\big) \right\}.$$

Similarly, the `ABSE-AS` problem is that of finding, for each edge $e = (a, b) \in E(T)$, a swap edge f^* such that:

$$f^* \in \arg\min_{f \in C(e)} \left\{ \lambda(f) := \frac{1}{|V(T_b)|} \sum_{v \in V(T_b)} \sigma_{G-e}\big(T_{e/f}, v\big) \right\}.$$

We will call $\mu(f)$ (resp., $\lambda(f)$) the *max-*(resp., *avg-*)*stretch* of f w.r.t. e.

4.2. The maximum stretch version

In this section we will show an efficient algorithm to solve the ABSE-MS problem in $O(mn + n^2 \log n)$ time. Notice that a brute-force approach would require $O(mn^2)$ time, given by the $O(n)$ time which is needed to evaluate the quality of each of the $O(m)$ swap edges, for each of the $n-1$ edges of T. Our algorithm will run through $n-1$ phases, each returning in $O(m + n \log n)$ time a BSE for a failing edge of T, as described in the following.

Let us fix $e = (a, b)$ as the failing edge. First, we compute in $O(m + n \log n)$ time all the distances in $G - e$ from s. Then, we filter the $O(m)$ potential swap edges to $O(n)$, i.e., at most one for each node v in T_b. Such a filtering is simply obtained by selecting, out of all edges $f = (x, v) \in C(e, \{v\})$, the one minimizing the measure $d_G(s, x) + w(f)$. Indeed, it is easy to see that the max-stretch of such selected swap edge is never worse than that of every other swap edge in $C(e)$. This filtering phase will cost $O(m)$ total time. As a consequence, we will henceforth assume that $|C(e)| = O(n)$.

Then, out of the obtained $O(n)$ swap edges for e, we further restrict our attention to a subset of $O(\log n)$ *candidates* as BSE, which are computed as follows. Let Λ denote a generic subtree of T_b, and assume that initially $\Lambda = T_b$. First of all, we compute in $O(|V(\Lambda)|)$ time a *centroid* c of Λ, namely a node whose removal from Λ splits Λ in a forest F of subtrees, each having at most $|V(\Lambda)|/2$ nodes [43]; then, out of all the swap edges, we select a candidate edge f minimizing the distance from s to c in $T_{e/f}$, i.e.,

$$f \in \arg \min_{(x',v') \in C(e)} \big\{ d_T(s, x') + w(x', v') + d_T(v', c) \big\};$$

then, we compute a *critical node* z for the selected swap edge f, i.e.,

$$z \in \arg \max_{z' \in V(T_b)} \sigma_{G-e}\big(T_{e/f}, z'\big).$$

We now select a suitable subtree Λ' of the forest F, and we pass to the selection of the next candidate BSE by recursing on Λ', until $|V(\Lambda')| = 1$. More precisely, Λ' is the first tree of F containing the first vertex of $V(\Lambda)$ that is encountered by following the path in T from z towards c (see Figure 1).

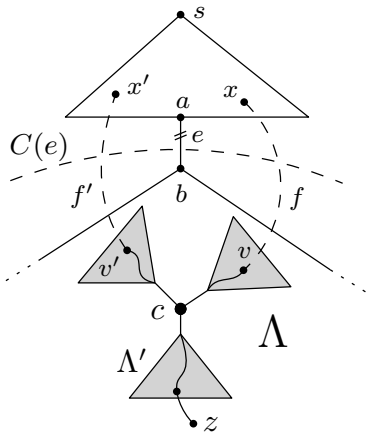

FIGURE 1. The situation illustrated in Lemma 19. The subtree Λ is represented by the three gray triangles along with the vertex c. $f = (x, v)$ is the candidate swap edge for e that minimizes $d_T(s, x) + w(f) + d_T(v, c)$, and z is its corresponding critical node. The algorithm will compute the next candidate swap edge by recursing on Λ'.

We start the candidate selection procedure with $\Lambda = T_b$ and we stop recursing whenever $|V(\Lambda)| = 1$. Due to the property of the centroid, the number of recursions will be $O\big(\log |V(T_b)|\big) = O(\log n)$, as promised, each costing $O(n)$ time. Moreover, at least one of the candidate edges will be a BSE for e, and hence it suffices to choose the edge minimizing the maximum stretch among the corresponding $O(\log n)$ candidate edges. This step is done within the recursive procedure by comparing the current candidate edge f with the best candidate resulting from the nested recursive calls.

A more formal description of each phase is shown in Algorithm 6. In the following we prove the correctness of our algorithm.

LEMMA 19. *Let $e = (a, b)$ be a failing edge, and let Λ be a subtree of T_b. Given a vertex $c \in V(\Lambda)$, let $f \in \arg\min_{(x',v') \in C(e)} \big\{d_T(s, v') + w(x', v') + d_T(v', c)\big\}$ and let z be a critical node for f. Let F be the forest obtained by removing the edges incident to c from Λ, and let Λ' be the tree of F containing the first vertex of the path from z to c in T that is also in $V(\Lambda)$. For any swap edge $f' \in C(e, V(\Lambda))$, if $\mu(f') < \mu(f)$ then $f' \in C(e, V(\Lambda'))$.*

Algorithm 2: ABSE-MS(e, Λ)

Input : a failing edge $e = (a, b) \in E(T)$, a subtree Λ of T_b.
Output: an edge $f \in C(e)$. If $C\big(e, V(\Lambda)\big)$ contains a BSE for e, f is a BSE for e.

$c \leftarrow$ Centroid of Λ;
Let $f \in \arg\min_{(x',v') \in C(e)} \big\{d_T(s, x') + w(x', v') + d_T(v', c)\big\}$;
if $|V(\Lambda)| = 1$ **then return** f;
Let $z \in \arg\max_{z' \in V(T_b)} \sigma_{G-e}\big(T_{e/f}, z'\big)$;
Let F be the forest obtained by removing the edges incident to c from Λ;
$y \leftarrow$ first vertex along the path from z towards c in T that is also in $V(\Lambda)$;
$\Lambda' \leftarrow$ tree of F containing y;
$f' \leftarrow$ ABSE-MS(e, Λ');
if $\mu(f') < \mu(f)$ **then return** f' **else return** f;

PROOF. Let $f = (x, v)$ and $f' = (x', v')$. We show that if $v' \in V(\Lambda) \setminus V(\Lambda')$ then $\mu(f') \geq \mu(f)$ (see also Figure 1). Indeed:

$$\begin{aligned}\mu(f') &\geq \sigma_{G-e}\big(T_{e/f'}, z\big) = \frac{d_{T_{e/f'}}(s, z)}{d_{G-e}(s, z)} = \frac{d_T(s, x') + w(f') + d_T(v', c) + d_T(c, z)}{d_{G-e}(s, z)} \\ &\geq \frac{d_T(s, x) + w(f) + d_T(v, c) + d_T(c, z)}{d_{G-e}(s, z)} \geq \frac{d_{T_e/f}(s, z)}{d_{G-e}(s, z)} = \sigma_{G-e}\big(T_{e/f}, z\big) \\ &= \mu(f),\end{aligned}$$

where we used the equality $d_T(v', z) = d_T(v', c) + d_T(c, z)$, which follows from the fact that the path from v' to z in T must traverse c as v' and z are in two different trees of F. □ □

LEMMA 20. *If $C\big(e, V(\Lambda)\big)$ contains a BSE for e then ABSE-MS(e, Λ) returns a BSE for e.*

PROOF. First of all notice that Algorithm 6 only returns edges in $C(e)$.

We prove the claim by induction on $|V(\Lambda)|$. If $|V(\Lambda)| = 1$ and $C\big(e, V(\Lambda)\big)$ contains a BSE f^* for e, then let f be the edge of $C(e)$ returned by Algorithm 6 and let $V(\Lambda) = \{c\}$. By choice of f, for every $v \in V(T_b)$,

$$d_{T_{e/f}}(s, v) \leq d_{T_{e/f}}(s, c) + d_T(c, v) = d_{T_{e/f^*}}(s, c) + d_T(c, v) = d_{T_{e/f^*}}(s, v),$$

from which we derive that $\mu(f) = \mu(f^*)$, and the claim follows.

If $|V(\Lambda)| > 1$ and $C\big(e, V(\Lambda)\big)$ contains a BSE for e, we distinguish two cases depending on whether the edge f computed by Algorithm 6 is a BSE for e or not. If that is the case, then $\mu(f) \leq \mu(f'') \forall f'' \in C(e)$ and the algorithm correctly returns f. Otherwise, by Lemma 19, any edge $f' \in C(e, V(\Lambda))$ such that $\mu(f') < \mu(f)$ must belong to $C\big(e, V(\Lambda')\big)$. It follows that Λ' contains a BSE for e and since $1 \leq |V(\Lambda')| < |V(\Lambda)|$ we have, by inductive hypothesis, that the edge f' returned by ABSE-MS(G, e, Λ') is a BSE for e. Clearly $\mu(f') < \mu(f)$ and hence Algorithm 6 correctly returns f'. □ □

Since each invocation of Algorithm 6 requires $O(n)$ time, Lemma 20 together with the previous discussions allows us to state the main theorem of this section:

THEOREM 3. *There exists an algorithm that solves the ABSE-MS problem in* $O(mn + n^2 \log n)$ *time.*

4.3. The average stretch version

In this section we show how the ABSE-AS problem can be solved efficiently in $O(mn \log \alpha(m, n))$ time. Before starting with the technical part, let us notice that such a problem asks, in general, for a different BSE as opposed to the resembling ABSE problem in which one seeks for a swap edge minimizing the *sum of all the distances* from the source of the nodes belonging to the detached subtree, which received a considerable attention in the literature. Our approach first of all, in a preprocessing phase, computes in $O(m\, n \log \alpha(m, n))$ time all the replacement shortest paths from the source after the failure of every edge of T [36]. Then, the algorithm will run through $n - 1$ phases, each returning in $O(m)$ time a BSE for a failing edge of T, as described in the following. Thus, the overall time complexity will be dominated by the preprocessing step.

Let us fix $e = (a, b) \in E(T)$ as the failing edge of T. The idea is to show that, after a $O(n)$ preprocessing time, we can compute the avg-stretch $\lambda(f)$ of any f in constant time. This immediately implies that we can compute a BSE for e by looking at all $O(m)$ swap edges for e.

Let $U = V(T_b)$ and let y be a node in U, we define:

$$M(y) = \sum_{v \in U} \frac{d_T(y,v)}{d_{G-e}(s,v)}$$

and

$$Q = \sum_{v \in U} \frac{1}{d_{G-e}(s,v)}.$$

Let $f = (x,y)$ be a candidate swap edge incident in $y \in U$. The avg-stretch of f can be rewritten as:

$$\lambda(f) = \sum_{v \in U} \frac{d_T(s,x) + w(f) + d_T(y,v)}{d_{G-e}(s,v)} = \big(d_T(s,x) + w(f)\big)Q + M(y).$$

Hence, the avg-stretch of f can be computed in $O(1)$ time, once Q and $M(y)$ are available in constant time. Observe that Q does not depend on y and can be computed in $O(n)$ time. The rest of this section is devoted to show how to compute $M(y)$ for every $y \in U$ in $O(n)$ overall time.

4.3.1. Computing $M(y)$ for all $y \in U$.

Let y e y' be two nodes in U such that y is a child of y' in T. Moreover, let $U_y = V(T_y)$, and let $Q_y = \sum_{v \in U_y} \frac{1}{d_{G-e}(s,y)}$. Hence, we can rewrite $M(y)$ and $M(y')$ as follows:

$$M(y) = \sum_{v \in U_y} \frac{d_T(y,v)}{d_{G-e}(s,v)} + \sum_{v \in U - U_y} \frac{w(y,y') + d_T(y',v)}{d_{G-e}(s,v)}$$

and

$$M(y') = \sum_{v \in U_y} \frac{w(y,y') + d_T(y,v)}{d_{G-e}(s,v)} + \sum_{v \in U - U_y} \frac{d_T(y',v)}{d_{G-e}(s,v)}.$$

Therefore, we have:

$$M(y) = M(y') + w(y,y')\big(-Q_y + (Q - Q_y)\big) = M(y') + w(y,y')\big(Q - 2Q_y\big). \tag{1}$$

The above equation implies that $M(y)$ can be computed in $O(1)$ time, once we have computed $M(y')$, Q and Q_y. As a consequence, we can compute all the $M(y)$'s as follows. First, we compute Q_y for every $y \in D$ in $O(n)$ overall time by means of a postorder visit of T_b. Notice also that $Q = Q_b$. Then, we compute $M(b)$ explicitly in $O(n)$ time. Finally, we compute all the other $M(y)$'s

by performing a preorder visit of T_b. When we visit a node y, we compute $M(y)$ in constant time using (1). Thus, the visit will take $O(n)$ time. We have proved the following:

THEOREM 4. *There exists an algorithm that solves the* ABSE-AS *problem in* $O(mn \log \alpha(m, n))$ *time.*

4.4. From best to good swap edges

Although the proposed solutions are quite efficient, their running time can become prohibitive for large and dense input graphs, since in this case they would amount to a time cubic in the number of vertices. Unfortunately, it turns out that their improvement is unlikely to be achieved, unless one could avoid the explicit recomputation of all post-failure distances from the source as they require the computation of all the new distances from the root for every possible edge failure of the given SPT. And in fact, as we will show in this thesis, the all-pairs shortest paths (APSP) problem can be reduced to such aforementioned computation. Thus, to improve our algorithms, one should refute the old-standing APSP conjecture, stating that, for general undirected and weighted graphs, there is no any $o(n^3)$ solving algorithm for it on a unit-cost RAM. To circumvent this problem, we then adopt a different approach, which by the way finds application for the (most relevant) max-stretch measure only: we renounce to optimality in the detection of a BSE, in return of a substantial improvement (in the order of a linear factor in n) in the runtime. More precisely, for such a measure, we will compute in an almost linear $O(m \log \alpha(m, n))$ time a set of *good* swap edges (GSE), each of which will guarantee a relative approximation factor on the maximum stretch of $3/2$ (tight) as opposed to that provided by the corresponding BSE. Moreover, a GSE will still guarantee an absolute maximum stretch factor w.r.t. the paths emanating from the source (in the surviving graph) equal to 3 (tight).

Besides that, we also point out another important feature concerned with the computation in a *distributed* setting of the GSE. Indeed, in [27] it was

shown that all our considered GSE can be computed in an *asynchronous message passing system* in $O(h)$ *ideal time*,[1] where h is the height (in terms of number of edges) of the input SPT, $O(n)$ message complexity, and $O(\delta_x)$ space complexity for each node x, where δ_x denotes the out-degree of x in the SPT. Thus, in a distributed environment, all the GSE can be basically computed in optimal space and time, as opposed to the recomputation of all the BSE, which presumably will ask for $O(h^2)$ ideal time, $O(n\,h)$ message complexity, and $O(h)$ space complexity for each node x, namely the long-standing bounds [34] for the very similar ABSE problem aiming to minimize the *average-distance* from the root.

4.5. The approximate solution for the maximum stretch version

In this section we show that for the max-stretch measure we can compute in an almost linear $O(m \log \alpha(m,n))$ time, a set of *good* swap edges (GSE), each of which guarantees a relative approximation factor on the maximum stretch of $3/2$ (tight), as opposed to that provided by the corresponding BSE. Moreover, as shown in the next section, each GSE still guarantees an absolute maximum stretch factor w.r.t. the paths emanating from the source (in the surviving graph) equal to 3 (tight).

LEMMA 21. *Let e be a failing edge in T, let*

$$g = (x,y) \in \arg \min_{(x',v') \in C(e)} \big\{ d_T(s,x') + w(x',v') \big\},$$

and, finally, let $f = (x',y')$ be a best swap edge for e w.r.t. ABSE-MS*. Then, $\mu(g)/\mu(f) \leq 3/2$.*

PROOF. Let z be the critical node for the good swap edge g, and let t (resp., t') denote the *least common ancestor* in T between y' and z (resp., y' and y). Let $D = d_T(s,x) + w(x,y) = d_{G-e}(s,y)$. By choice of g, it holds that $d_{G-e}(s,z) \geq D$ and $d_{G-e}(s,y') \geq D$. We divide the proof into the following two cases, as depicted in Figure 2: either (1) t is an ancestor of t' in T, or (2) t' is an ancestor

[1]This is the time obtained with the ideal assumption that the communication time for each message to a neighboring process takes constant time, as in the synchronous model.

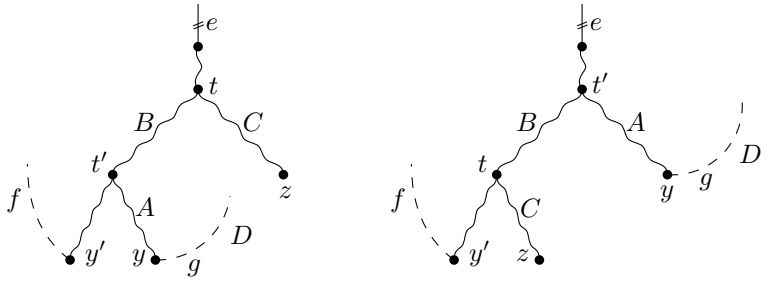

FIGURE 2. The figure shows the two cases of the analysis, on the left t is an ancestor of t', while on the right the opposite holds. The splines denote a path, while the straight lines represent a single edge.

of t in T. Let A, B, C denote the distance in T between y and t', t' and t, t and z, respectively.

Case 1. Since t is an ancestor of t' (left side of Figure 2), we have that $d_{T_{e/f}}(s, y) \geq D + A$ and we can write:

$$\sigma_{G-e}(T_{e/f}, y) \geq \frac{D+A}{d_{G-e}(s,y)} = \frac{D+A}{D} \geq \frac{D+A}{d_{G-e}(s,z)},$$

and similarly $\sigma_{G-e}(T_{e/f}, z) \geq \frac{D+B+C}{d_{G-e}(s,z)}$. Moreover, by the definition of $\mu(\cdot)$ we have that $\mu(f) \geq \max\{\sigma_{G-e}(T_{e/f}, y), \sigma_{G-e}(T_{e/f}, z)\}$. The previous inequalities together imply:

$$\frac{\mu(g)}{\mu(f)} \leq \frac{\sigma_{G-e}(T_{e/g}, z)}{\max\{\sigma_{G-e}(T_{e/f}, y), \sigma_{G-e}(T_{e/f}, z)\}} \leq \frac{A+B+C+D}{D + \max\{A, B+C\}}. \tag{2}$$

Now we divide the proof into two subcases, depending on whether $B + C \geq A$ or $B + C < A$. Observe that $D \geq d_G(s, y) \geq A$. If $B + C \geq A$, then (2) becomes:

$$\frac{\mu(g)}{\mu(f)} \leq \frac{A+B+C+D}{B+C+D} = 1 + \frac{A}{B+C+D} \leq 1 + \frac{A}{2A} = \frac{3}{2},$$

otherwise, if $B + C < A$, then (2) becomes:

$$\frac{\mu(g)}{\mu(f)} \leq \frac{A+B+C+D}{A+D} < \frac{2A+D}{A+D} = 1 + \frac{A}{A+D} \leq 1 + \frac{A}{2A} = \frac{3}{2}.$$

Case 2. Assume now that t' is an ancestor of t (right side of Figure 2). Since

$$\mu(f) \geq \sigma_{G-e}(T_{e/f}, y) \geq \frac{d_{G-e}(s, y') + A + B}{d_{G-e}(s, y)} = \frac{d_{G-e}(s, y') + A + B}{D},$$

we have that:

$$\begin{aligned}\frac{\mu(g)}{\mu(f)} &\leq \frac{A+B+C+D}{d_{G-e}(s,z)} \cdot \frac{D}{d_{G-e}(s,y')+A+B} \\ &\leq \frac{A+B+C+D}{d_{G-e}(s,z)} \cdot \frac{D}{A+B+D}\end{aligned}$$

and since $d_{G-e}(s,z) \geq d_G(s,z) \geq C$, and recalling that $d_{G-e}(s,z) \geq D$, we have:

$$\frac{\mu(g)}{\mu(f)} \leq \frac{A+B+C+D}{A+B+D} \cdot \frac{D}{\max\{C,D\}} = \left(1+\frac{C}{A+B+D}\right) \cdot \frac{D}{\max\{C,D\}}. \tag{3}$$

Moreover, notice that also the following holds:

$$\begin{aligned}\frac{\mu(g)}{\mu(f)} &\leq \frac{\mu(g)}{\sigma_{G-e}(T_{e/f},z)} \leq \frac{A+B+C+D}{d_{G-e}(s,z)} \cdot \frac{d_{G-e}(s,z)}{d_{G-e}(s,y')+d_T(y',t)+C} \\ &\leq \frac{A+B+C+D}{C+D} = 1+\frac{A+B}{C+D}. \qquad (4)\end{aligned}$$

We divide the proof into the following two subcases, depending on whether $D \geq C$ or $D < C$. In the first subcase, i.e., $D \geq C$, we have that (3) becomes $\frac{\mu(g)}{\mu(f)} \leq 1+\frac{C}{A+B+D}$, and hence, by combining this inequality with (4), we obtain:

$$\begin{aligned}\frac{\mu(g)}{\mu(f)} &\leq 1+\min\left\{\frac{C}{A+B+D}, \frac{A+B}{C+D}\right\} \\ &\leq 1+\min\left\{\frac{C}{A+B+C}, \frac{A+B}{2C}\right\} \leq 1+\frac{1}{2} = \frac{3}{2}.\end{aligned}$$

In the second subcase, i.e., $D < C$, (3) becomes:

$$\frac{\mu(g)}{\mu(f)} \leq \left(1+\frac{C}{A+B+D}\right) \cdot \frac{D}{C} \leq \frac{D}{C} + \frac{D}{A+B+D} < 1+\frac{D}{A+B+D}, \tag{5}$$

and hence, by combining (5) and (4), we have that:

$$\begin{aligned}\frac{\mu(g)}{\mu(f)} &\leq 1+\min\left\{\frac{D}{A+B+D}, \frac{A+B}{C+D}\right\} \\ &\leq 1+\min\left\{\frac{D}{A+B+D}, \frac{A+B}{2D}\right\} \leq 1+\frac{1}{2} = \frac{3}{2},\end{aligned}$$

from which the claim follows. □ □

Given the result of Lemma 21, we can derive an efficient algorithm to compute all the GSE for ABSE-MS. More precisely, in [47] it was shown how to find them in $O(m\,\alpha(m,n))$ time. Essentially, the approach used in [47] was based on a reduction to the *SPT sensitivity analysis* problem [54]. However, in [49] it

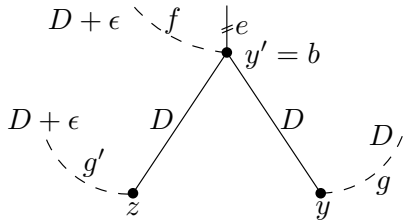

FIGURE 3. A tight example showing that the quality of the good swap edge g computed by the algorithm is a factor of $3/2$ away from the qualify of a best swap edge f. In the picture, it is assumed that the distance from s to b is equal to 0, while the three dashed edges are assumed to be incident to the source, and the labels correspond to their weight. Then, $\mu(f) = \sigma_{G-e}(T_{e/f}, y) = \frac{2D+\epsilon}{D} \simeq 2$, while $\mu(g) = \sigma_{G-e}(T_{e/g}, z) = \frac{3D}{D+\epsilon} \simeq 3$, for small values of ϵ.

was proposed a faster solution to such a problem, running in $O(m \log \alpha(m,n))$ time. Thus, we can provide the following

THEOREM 5. *There exists a $3/2$-approximation algorithm that solves the ABSE-MS problem in $O(m \log \alpha(m,n))$ time.*

We conclude this section with a tight example which shows that the analysis provided in Lemma 21 is tight (see Figure 3).

4.6. Quality analysis

As for previous studies on swap edges, it is interesting now to see how the tree obtained from swapping a failing edge $e = (a,b)$ with its BSE f compares with a true SPT of $G - e$. According to our swap criteria, we will then analyze the lower and upper bounds of the max- and avg-stretch of f, i.e., $\mu(f)$ and $\lambda(f)$, respectively.

As already observed in the introduction, it is well-known [47] that for the swap edge, say g, which belongs to the shortest path in $G-e$ between s and the root of the detached subtree T_b, we have that for any $v \in V(T_b)$, $\sigma_{G-e}(T_{e/g}, v) \leq 3$. This immediately implies that $\mu(g), \lambda(g) \leq 3$, namely $\mu(f), \lambda(f) \leq 3$. These bounds happen to be tight, as shown in Figure 4.

Let us now analyze the lower and upper bounds of the max-stretch of a *good* swap edge g, i.e., $\mu(g)$, as defined in the previous section. First of all, once again it was proven in [46] that for any $v \in V(T_b)$, $\sigma_{G-e}(T_{e/g}, v) \leq 3$, which

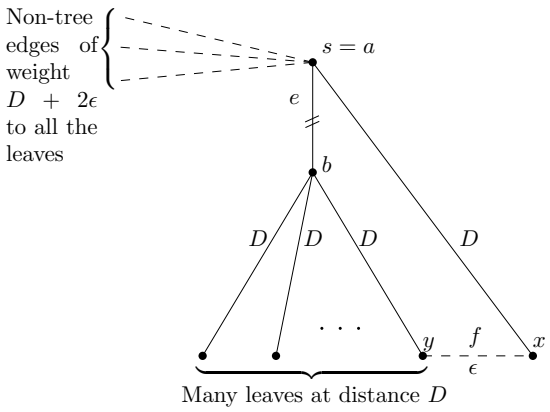

FIGURE 4. Tight ratios for $\mu(f)$ and $\lambda(f)$. In the picture, the SPT T (solid edges) along with the removed edge $e = (a, b)$ of weight 0; non-tree edges are dashed and the best swap edge (for both the maximum and the average stretch) is easily seen to be $f = (x, y)$, of weight $\epsilon > 0$. Then, we have that $\mu(f) = \frac{3D+\epsilon}{D+2\epsilon}$, which tends to 3 for small values of ϵ, while $\lambda(f)$ tends to 3 as well, as soon as the number of leaves grows. Notice that f is also a good swap edge for e.

implies that $\mu(g) \leq 3$. Moreover, the example shown in Figure 4 can be used to verify that this bound is tight.

CHAPTER 5

Conclusion

Regarding the ABSE tree spanner problems, we provided two efficient linear-space solutions for both the weighted and the unweighted version, running in $O(n^2 \log^4 n)$ time and $O(n^2 + m \log^2 n)$ space, respectively.

Our future research on this problem will follow several directions. One of them will be that of analyzing the *quality* of the swap tree spanner as compared to that of an optimal tree spanner of the graph deprived of the failed edge. To address this task, one should evaluate the ratio between the stretch factors of the two trees. A similar study was performed in [26], where the authors focused on unweighted graphs, and showed that if stretches are measured w.r.t. distances in G (i.e., according to their criteria, but differently from our intent which is that of measuring stretches in the affected graph), and if moreover the initial tree is an *optimal* tree spanner, then this ratio is bounded by 2, and this is tight. Unfortunately, computing an optimal tree spanner is hard, and it is unknown what will be the outcome when the initial tree is suboptimal. However, we conjecture that a suitable preprocessing of the initial tree could avoid this pathological behavior. Finally, we also plan to focus our attention on the related problem of handling single node failures.

Regarding the ABSE on a SPT, we have studied two natural swap problems, aiming to minimize, after the failure of any edge of the given SPT, either the maximum or the average stretch factor induced by a swap edge. We have first proposed two efficient algorithms to solve both problems. Then, aiming to the design of faster algorithms, we developed for the maximum-stretch measure an almost linear time algorithm guaranteeing a $3/2$-approximation w.r.t. the optimum.

Concerning future research directions, the most important open problem remains that of finding a linear-size edge-fault-tolerant SPT with a (maximum)

stretch factor w.r.t. the root better than 3, or to prove that this is unfeasible. Another interesting open problem is that of improving the running time of our exact solutions. Notice that both our exact algorithms pass through the computation of all the post-failure single-source distances, and if we could avoid that we would get faster solutions. At a first glance, this sounds very hard, since the stretches are heavily dependant on post-failure distances, but, at least in principle, one could exploit some monotonicity property among swap edges that could allow to skip such a bottleneck. Besides that, it would be nice to design a fast approximation algorithm for the average-stretch measure. Apparently, in this case it is not easy to adopt an approach based on good swap edges as for the maximum-stretch case, since swap edges optimizing other reasonable swap criteria (e.g., minimizing the distance towards the root of the detached subtree, or minimizing the distance towards a detached node) are easily seen to produce an approximation ratio of 3 as opposed to a BSE. A candidate solution may be that of selecting a BSE w.r.t. the sum-of-distances criterium, which can be solved in almost linear time [29], but for which we are currently unable to provide a corresponding comparative analysis.

Finally, we mention that a concrete task which will be pursued is that of conducting an extensive experimental analysis of the true performances of our algorithms, to check whether for real-world instances the obtained stretches are sensibly better or not w.r.t. the theoretical bounds.

Part 2

Labeling Problems

CHAPTER 6

Introduction

A *shortest-path query* asks the shortest path between two vertices in a graph. Without doubt, answering to this kind of queries is one of the most fundamental operations on graphs, as it has a wide range of applications, e.g. social networks analysis [55], route planning in road networks [28, 25], intelligent transport systems [18], routing in communication networks [22, 24]. Baseline methods, e.g. Breadth First Search or Dijkstra's algorithm, yield unsustainable *query times* to answer such requests in graphs arising from the mentioned applications, which tend to be huge.

To overcome this limit, tens of smarter approaches have been proposed in the last decade [4, 25] which adopt the common strategy of preprocessing the graph in advance to speed-up the query phase. Among them, the *2-hop cover distance/path-reporting labeling scheme* (2–HCL from now on) is based on the idea of representing the shortest paths of a graph as the concatenation at an intermediate so-called *hub* vertex of the two shortest paths emanating from the corresponding end vertices.

Roughly speaking, the label at each vertex will contain the length and the next-hop vertex of a shortest path towards each hub vertex, and to retrieve a shortest path between two vertices it will suffice to join their labels and find the common hub that minimizes the sum of the two distances. The difficult point here is to find a small set of hub vertices covering a shortest path for each pair of vertices in the graph, since in this way the corresponding labeling will be compact. Indeed, finding a minimum-size set of hub vertices is known to be NP-hard [19].

6.1. Related Work

In the practice, 2–HCL is currently considered the best labeling scheme since:

- It is general: it can be applied on all kinds of graphs, including directed weighted ones [23, 28];
- Even in networks with tens of billions of vertices, it combines extremely low query times with affordable space overhead and reasonable preprocessing effort [2];
- It is suited to be used in distributed settings [57];
- Several practical approximation algorithms are known [19] - among them, the recent *pruned landmark labeling* (PLL) [2] achieves considerably better scalability than other methods.

A further level of complexity, when dealing with this kind of approaches, is represented by the fact that, in general, real-world networks are prone to failures. In this case, in fact, if a failing edge is on a queried shortest path, then a distance query will return an underestimated values w.r.t. a true shortest path in the damaged graph, while a path-reporting query would actually return an unfeasible path!

Since reprocessing the graph from scratch after each failure is not a reasonable recovery strategy (see [23]), devising a *fault-tolerant* scheme, is the most viable option. This is usually achieved by suitably *enriching* the underlying data structure and by accordingly modify the query strategy to consider such enrichment.

However, if the paths to be reported are constrained to be *shortest* also in presence of failures, then this might result in storing an unpractical amount of additional information [51]. In this case, a reasonable compromise is that of relaxing the optimality constraint and devise more compact schemes that return *approximate* distances (shortest paths, resp.), i.e. distances (paths, resp.) that are guaranteed to be longer than the corresponding true distances (shortest paths, resp.) by at most a given *stretch factor*, for any possible kind of failure that has to be handled.

Examples of this strategy in the literature are various [21, 10], where many structures have been studied in the fault-tolerant approximate setting. However, to the best of our knowledge, no previous work has ever investigated 2–HCL in such a setting.

6.2. Our Results

In this thesis, we move forward in this direction and present a simple and efficient way of enriching a 2–HCL in order to make it *resilient* to the failure of a subset of any k edges.

In more details, given an n-vertex and m-edge graph G, we show that such an enrichment can be done by exploiting the notion of so-called *edge-independent* spanning trees. More precisely:

- For $k = 1$ and G at least $(k+1)$-edge connected, the enrichment takes $O(m+n)$ additional time and $O(n)$ additional space;
- For $k = 2$ and G at least $(k+1)$-edge connected, the enrichment takes $O(n^2)$ additional time and $O(n)$ additional space;
- For $k = 3$ and G at least $(k+1)$-edge connected, the enrichment takes $O(n^3)$ additional time and $O(n)$ additional space;
- For $k > 3$ and G at least $(2k+2)$-edge connected, the enrichment takes $O(k^2n^2)$ additional time and $O(kn)$ additional space.

Even though the above solutions have in principle replacement paths which can be linearly (in n and k) stretched (as compared to new shortest paths in the surviving graph), in practice the situation is much better. To assess that, we provide an experimental evaluation, conducted for the case $k = 1$, showing the quality of our approach in terms of stretch, space requirements and preprocessing/query time. Our data shows the new method to be really effective and competitive when compared with the performance of the benchmark non-fault-tolerant scheme, equipped with a point-of-failure rerouting policy. Notice that our results can be extended to weighted graphs as well, and that our solutions are able to answer to the most general path-reporting queries.

CHAPTER 7

Fault-Tolerant Labelings

7.1. Background

In what follows, we provide the notation and the basic definitions that are used throughout the thesis. Given an unweighted undirected graph $G = (V, E)$, we denote by π^G_{xy} a shortest path between two vertices $x, y \in V$, and by d^G_{xy} (or $|\pi^G_{xy}|$) its length (a.k.a. the *distance* between x and y in G).

In addition, given a shortest path π^G_{xy} we denote by $p(x, \pi^G_{xy})$ ($p(y, \pi^G_{xy})$, resp.) the *predecessor* of x (y, resp.) within π^G_{xy}, i.e. the vertex v such that $(x, v) \in \pi^G_{xy}$ ($(v, y) \in \pi^G_{xy}$, resp.). Given two simple paths a and b, we denote by $p = a \oplus b$ the path p obtained by concatenating them.

Furthermore, we say that a graph G is k *edge connected* if $|E| > k$ and there does not exist a set of edges, of cardinality at most $k - 1$, whose removal disconnects the graph. Given a graph G and a distinguished *root* vertex $r \in V$, then $\texttt{IT} = \{T_1, T_2, \ldots, T_q\}$ is a collection of q *edge-independent* spanning trees of G if and only if, for each vertex $v \in V$, and for each $i \neq j$, $\pi^{T_i}_{rv}$ and $\pi^{T_j}_{rv}$ are pairwise edge-disjoint paths, i.e. they do not share any edge [39].

In what follows, we give a slightly modified definition of 2–HCL, suited for answering to path-reporting queries, inspired by that of [19] for distance queries.

DEFINITION 7 (2–HCL). *Let $G = (V, E)$ be an undirected graph. Let the* path label *$P(v)$ of a vertex $v \in V$ be a set of tuples of the form $\langle u, d^G_{uv}, p(v, \pi^G_{uv}))\rangle$ and let $P(G) = \{P(v)\}_{v \in V}$ (denoted as P when the graph is clear from the context) be a collection of such labels. Let a path query operation on $P(G)$ from a vertex s to a vertex t be defined as:*

$$\textsc{pQuery}(s, t, P) = \begin{cases} \langle h, d^G_{hs}, p(s, \pi^G_{hs})\rangle \mid h = \underset{v \in P(s) \cap P(t)}{\operatorname{argmin}} \{d^G_{vs} + d^G_{vt}\} & \text{if } P(s) \cap P(t) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

where, for the sake of brevity, we denote by $P(s) \cap P(t)$ *the set of vertices such that* $\{v | \langle v, d^G_{vs}, p(v, \pi^G_{vs})\rangle \in P(s) \wedge \langle v, d^G_{vt}, p(v, \pi^G_{vt})\rangle \in P(t)\}$. *Then,* $P(G)$ *is a 2–HCL of* G *if and only if, for any pair of vertices* s *and* t *of* G *it satisfies the cover property [19], i.e., if* s *and* t *are connected in* G *and* h *is the first argument returned by* PQUERY(s, t, P)*, then* $d^G_{st} = d^G_{hs} + d^G_{ht}$ *and* h *is called a* hub *that* covers s *and* t*, otherwise* PQUERY$(s, t, P) = \emptyset$.

Given the above, it is clear that a 2–HCL $P(G)$ of a graph G can be used also to retrieve an entire shortest path π^G_{st} (i.e. the corresponding set of vertices/edges) for any given pair of vertices s, t of G by the procedure shown in Algorithm 3. The routine essentially builds incrementally two shortest paths, one from s toward a hub h and the other from t toward the same hub h, and eventually combines them into a path from s to t which is exactly the shortest between s and t, by the cover property. The two paths are computed by the repeated application of the above path query, i.e. by concatenating edges connecting vertices to the corresponding predecessors. For the sake of clarity, we represent the shortest path as a doubly linked list of edges.

Algorithm 3: Computation of a shortest path via $P(G)$.

Input : $P(G)$, a pair of vertices x and y of G.
Output: A shortest path $p = \pi^G_{xy}$ between x and y.

```
1  p ← ∅; p_x ← ∅; p_y ← ∅; c_1 ← x; c_2 ← y;
2  if c_1 = c_2 then return p;
3  l_1 ← PQUERY(c_1, c_2, P);
4  l_2 ← PQUERY(c_2, c_1, P);
5  h ← l_1.first                    /* h equals l_2.first by construction */
6  while c_1 ≠ l_1.first ∨ c_2 ≠ l_2.first do
7      if c_1 ≠ l_1.first then
8          p_x.pushFront((c_1, l_1.third));
9          c_1 ← l_1.third;
10         l_1 ← PQUERY(c_1, h, P);
11     if c_2 ≠ v then
12         p_y.pushBack((c_2, l_2.third));
13         c_2 ← l_2.third;
14         l_2 ←← PQUERY(c_2, h, P);
15 p = p_x ⊕ p_y;
16 return p
```

7.1.1. Problem Statement.

Let $G = (V, E)$ be an unweighted and undirected graph. We aim at computing what we call a *k-edge-fault-tolerant path-reporting labeling scheme* (k-EFTPL, in short), i.e. a labeling $P(G, k)$ that, for any pair of vertices s, t of G, can be used to retrieve: i) a shortest path π_{st}^{G} between s and t in G; ii) a *backup path* γ_{st}^{G-F} between s and t, i.e. a simple path (if any) connecting s and t in $G - F$ for any $F \subseteq E$ such that $|F| \leq k$.

The stretch factor of a k-EFTPL $P(G, k)$ is defined as the maximum ratio, for any pair of vertices s, t and for any $F \subseteq E$ such that $|F| \leq k$, between the length of a backup path and that of the corresponding shortest path in $G - F$, i.e.: $\alpha(P(G,k)) = \max_{s,t \in V, \forall F \subseteq E : |F| \leq k} \frac{|\gamma_{st}^{G-F}|}{|\pi_{st}^{G-F}|}$. Note that, whenever π_{st}^{G-F} does not exist (i.e. s and t are disconnected in $G - F$), we assume $\frac{|\gamma_{st}^{G-F}|}{|\pi_{st}^{G-F}|}$ to be 1. A k-EFTPL $P(G, k)$ such that $\alpha(P(G, k)) = 1$ is called *exact* while is called *approximate* otherwise.

7.2. A linear-stretch approximate k-EFTPL

In this section, we show how to build an approximate k-EFTPL $P(G, k)$ for a given input graph G. Our approach essentially consists in a suitable enrichment of a 2–HCL $P(G)$, based on the use of edge-independent trees, as follows. Given G, we start by computing $P(G)$ via a standard method, e.g. PLL [2] and, to be able to "resist" to the failure of k edges, we compute $k+1$ edge-independent trees $T_1, T_2, \ldots T_{k+1}$ of G and root them all at a same distinguished vertex, say $r \in V$. Note that, such trees guarantess that, should any k edges $e_1, e_2, \ldots, e_k$ of the graph fail, for any vertex $v \in V$ there will exist (at least) one value $i \in [1, k + 1]$ such that v and r are connected, in T_i, by a simple path not containing any of the $e_1, e_2, \ldots, e_k$ edges. This can be trivially proved by contradiction, by elaborating on the well-known results from graph theory, summarized in the following.

THEOREM 6 ([45]). *Let* IT *be a graph obtained by merging a collection of $q + 1$ edge-independent spanning trees $\{T_i\}_{i=1,2,\ldots,q+1}$ of a graph G. Then,* IT *is $(q + 1)$-edge connected.*

Hence, we use such rooted trees to build a set of what we call *tree labels* that are assigned to the vertices in order to allow the retrieval of a backup path in presence of k edge failures. Intuitively, such labels store the predecessor of a vertex within a shortest path toward the root r, for each considered tree. Clearly, a necessary condition to guarantee that any pair of vertices remain connected even in presence of k edge failures is that G is $(k+1)$-edge connected. Our approach is based on the following general consideration: if G admits $k+1$ independent trees $T_1, T_2, \dots T_{k+1}$, all rooted at a same distinguished vertex, say $r \in V$, and we are able to compute them.

Depending on the value of k, we make use of different approaches to build the $k+1$ edge-independent trees. Clearly, a necessary condition to guarantee that any pair of vertices remains connected even in presence of k edge failures is that G is at least $(k+1)$-edge connected. Then, if $k \in \{1, 2, 3\}$ and G is $(k+1)$-edge connected, we can compute $k+1$ edge-independent trees in polynomial time, with a time complexity of $O(m+n)$, $O(n^2)$ and $O(n^3)$, respectively [39, 17, 20].

On the other hand, if $k > 3$ and G is h-edge connected, with $k+1 \leq h \leq 2k+1$, then to the best of our knowledge it is not possible to build $k+1$ edge-independent trees in polynomial time. However, if G is at least $(2k+2)$-edge connected, then we can build $k+1$ *edge-disjoint* spanning trees of G, which are clearly also edge-independent, by using the approach proposed in [52], running in $O(k^2 n^2)$ time.

From now on, we will suppose to have at our disposal $k+1$ edge-independent trees of G, built as aforementioned. Hence, for each vertex, there will always be a tree label containing a viable predecessor to be used to reach r, even in presence of k arbitrary edge failures in G. More in details, given a vertex $v \in V$ and an independent tree T_i of G (rooted at a vertex $r \in V$), we define a *tree label entry* to be a pair $\langle r, p(v, \pi_{vr}^{T_i}) \rangle$. Such pair essentially encodes for v a path to follow toward the root r of the i-th tree in the form of a predecessor within the tree. Tree label entries are stored within what we call a *tree label* $T(v,k)$,[1] for

[1]The k parameter stands for the ability of resisting to k edge failures by $k+1$ trees.

any $v \in V$. Hence, any vertex is assigned two labels, namely $P(v)$ and $T(v,k)$, to be used in combination in order to be tolerant to the failure of k edges.

We call the set $\{T(v,k)\}_{v \in V}$ a *k-tree labeling* of G (denoted as $T(G,k)$ or T when clear from the context) while we define $P(G,k)$ to be the union of $P(G)$ and $T(G,k)$. Moreover, we call the chosen r the *root* of $P(G,k)$.

The routine for building the k-tree labeling of a graph G, and therefore $P(G,k)$, is summarized in Algorithm 4. We represent $T(v,k)$ again as a doubly linked list, since it will be convenient for defining a suited query algorithm. Now, we can easily bound the computational complexity of Algorithm 4 as follows.

Algorithm 4: Building a k-EFTPL $P(G,k)$ of a graph $G=(V,E)$

Input : Graph $G=(V,E)$, parameter $k \geq 1$, a 2–HCL $P(G)$ of G
Output: A k-EFTPL $P(G,k)$ of G

1 Compute $k+1$ edge-independent trees $T_1, T_2, \ldots T_{k+1}$;
2 Choose a distinguished vertex $r \in V$, root all k trees in r;
3 $T(G,k) \leftarrow \emptyset$;
4 **foreach** $v \in V$ **do**
5 | **foreach** $i = 1,2,\ldots,k+1$ **do** $T(v,k).push\_back(\langle r, p(v, \pi_{vr}^{T_i})\rangle)$;
6 | $T(G,k) \leftarrow T(v,k)$;
7 $P(G,k) = P(G) \cup T(G,k)$;
8 **return** $P(G,k)$

LEMMA 22. *Algorithm 4 takes $O(\Delta + k|V|)$ time, where Δ is the worst-case time for computing $k+1$ edge-independent trees.*

PROOF. Clearly, performing line 1 takes $O(\Delta)$ while the execution of the lines 2-3 takes constant time. Moreover, the outer loop of line 4 is executed exactly $|V|$ times while the inner loop of line 5 is performed exactly $k+1$ times. Since the operation of line 5 can be done in $O(1)$ time,[2] the claim follows. □

LEMMA 23. *Algorithm 4 builds a k-EFTPL $P(G,k)$ whose overall size, in terms of label entries, is $O(|P(G)| + k|V|)$.*

PROOF. To prove the statement, it is enough to bound the size of $T(G,k)$, since the size of $P(G,k)$ is trivially upper bounded by the sum of the sizes of

[2] By storing each T_i as an array of predecessors.

$P(G)$ and $T(G,k)$. At line 3, $T(G,k)$ is empty, and exactly one entry per value of i is added to the tree label $T(v,k)$ of each vertex $v \in V$ at line 5. Since i ranges from 1 to k, the claim follows. □

Given the construction of Algorithm 4, we now provide a query algorithm that can be used for either retrieving a true shortest path, when the graph is not subject to failures, or a backup path otherwise.

The procedure, summarized in Algorithm 5, roughly works as follows. Given $P(G,k)$ and a pair of vertices s and t, the routine starts by trying to retrieve the true shortest path, by relying on $P(G)$ and on the corresponding PQUERY strategy. As in the previous case, the shortest path is built incrementally and by combining two separate parts. If no failures are currently present in the network, then the algorithm essentially coincides with Algorithm 3 and returns the shortest path. Otherwise, if a number $k' < k+1$ of failures are occurring on G, then two cases can occur: either the shortest path from s to t includes at least one of such failures or not. To determine that, the algorithm needs essentially to check, at every step of the construction, whether the edge leading to the predecessor on the original shortest path has failed or not. In the negative case, clearly, the algorithm proceeds as nothing happened, i.e. as Algorithm 3, by repeating the application of PQUERY. In the affirmative case, instead, the algorithm starts looking for backup paths by using available tree label entries, which essentially encode edges connecting to predecessors in the independent trees (see Lines 4-9 of Algorithm 6). Since a number of failures can affect different trees, tree label entries are scanned sequentially until one non-failed edge if found. Again two sub-paths are built and then, eventually, combined.

Note that, in this case, the two sub-paths might be not connected since one of the two searches might end up in the original hub, while the other in the root, and viceversa. This happens when one of the two sub-paths reaches the original hub while the other does not. In this case, Algorithm 7 takes care of finding the missing part. In general, we are able to prove the following.

LEMMA 24. *Algorithm 5 always computes a backup path connecting x to y, for any pair $x, y \in V$.*

Algorithm 5: Query algorithm for a k-EFTPL $P(G,k)$.

Input : $P(G,k)$, two vertices x and y.
Output: A simple path p between x and y.

$p \leftarrow \emptyset$; $p_x \leftarrow \emptyset$; $p_y \leftarrow \emptyset$; $c_1 \leftarrow x$; $c_2 \leftarrow y$;
$l_0 \leftarrow$ the 0-th tree label of $T(z,k)$; /* Any tree label entry, first field is always r */
$r \leftarrow l_0.first$;
if $x = y$ **then return** p;
$l_1 \leftarrow$ PQUERY(c_1, c_2, P);
$h \leftarrow l_1.first$ /* h equals $l_2.first$ by construction */
$p_x \leftarrow$ NAVIGATEFROM(x,h,r); $p_y \leftarrow$ NAVIGATEFROM(y,h,r);
if $p_x.tail() \neq p_y.tail()$ **then**
  if $p_x.tail() \neq h$ **then** FORCETOROOT(p_x, r);
  else FORCETOROOT(p_y, r);
else $p \leftarrow p_x \oplus reverse(p_y)$;
for $e \in p$ **do**
  if $|e| > 1$ **then** $p \leftarrow p \setminus e$ /* Removes any duplicates */ ;
return p;

Algorithm 6: Procedure NAVIGATEFROM(z,h,r)

Input : $P(G,k)$, a vertex z, its hub h, the root of the trees r.
Output: A simple path p_z connecting z to either h or r.

$p_z \leftarrow \emptyset$; FAILED $\leftarrow false$; $i \leftarrow 0$; /* i-th edge-independent tree in use */
$l_z \leftarrow$ PQUERY(z,h,P);
while $z \neq h \wedge z \neq r$ **do**
  if FAILED $= true$ **then** /* failure previously encountered, keep going toward r */
    Let l_i be the i-th tree label of $T(z,k)$;
    if $(z, l_i.second) \in E$ **then**
      $p_z.pushFront((z, l_i.second))$;
      $z \leftarrow l_i.second$;
    else $i \leftarrow i + 1$; /* change tree */
  else
    if $(z, l_z.third) \notin E$ **then** FAILED $\leftarrow true$; /* failure encountered */
    else /* keep navigating toward h */
      $p_z.pushFront((z, l_z.third))$;
      $z \leftarrow l_z.third$;
      $l_z \leftarrow$ PQUERY(z,h,P);
return p_z;

PROOF. Algorithm 5 makes use of two other algorithms, i.e.: Algorithm 6 and Algorithm 7. The correctness of Algorithm 6 comes directly from the correctness of the function PQUERY(x, h, P), due to the fact that it will iteratively build a path by calling it. Since we have built a 2-EFTPL it will be always possible to find a path between x and h using P. We need to show that we are able to find a path between any two pair $x, y \in V$ even in presence of any failure. If Algorithm 7 is called it means that a failed edge has been found during the navigation towards the hub, therefore a new procedure needs to be used. In this case we will force the path to reach the root of the trees. Summarizing, we will have 2 cases: if two endpoints are the same, then the algorithm terminates, they can be either on the hub, or on the root of the trees. Otherwise, if the two endpoints are different, we force the endpoint which is equal to the hub, to go to the root of the trees instead. If such the case, a path will always exists due to the connectivity properties of P, and it will be found by Algorithm 7 using PQUERY(x, h, P). □

THEOREM 7. *Algorithm 5 builds a k-EFTPL $P(G, k)$ with $\alpha(P(G, k)) \in O(kn)$.*

PROOF. The length of any path found by Algorithm 5 from any node v to the root r is always upper bounded by $\text{DIAM}_{max} = \max_{i=1,\dots,k+1} 2 \cdot \text{DIAM}(T_i)$, where $\text{DIAM}(T_i)$ is the diameter of the i-th edge-independent tree T_i. Since we could find a failed edge up to k times (one for each tree T_i), we could have to pay $\text{DIAM}(T_i)$ each time. Therefore we could have $k \cdot \text{DIAM}(T_i)$, hence, the claim follows. □

Algorithm 7: Procedure FORCETOROOT(p, r).

Input : $P(G, k)$, a path p, a vertex r.

```
1 i ← 0; z ← p.tail();
2 while z ≠ r do
3 |   Let l_i be the i-th tree label of T(z, k);
4 |   if (z, l_i.second) ∈ E then
5 |   |   p.pushFront((z, l_i.second));
6 |   |   z ← l_i.second;
7 |   else i ← i + 1;                          /* change tree */
```

7.3. Experimental evaluation

In this section we provide an experimental evaluation of our approach. In particular, we implemented, in C++:

(1) The PLL approach of [2] for building 2–HCL;

(2) Algorithm 4 for building an approximate k-EFTPL, for $k = 1$.

We embedded all our code within *NetworKit*[3], a well-known open-source framework for large-scale network analysis. Then, to assess the performance of our method, we established the following experimental process.

First, inspired by other papers on the subject (e.g. [51, 23]), we selected a meaningful set of real-world networks of various types (including road graphs, peer to peer networks, and AS communication networks[4]) and treated them as undirected, unweighted graphs. Details about inputs are reported in Table 1 where we show, for each network, the corresponding type, and the number of vertices and edges of the largest 2-edge connected component (second, third and fourth columns, resp.).

Network	**Type**	$\|V\|$	$\|E\|$	**time** (seconds)		**space per vertex** (bytes)	
				PLL	KHL	$P(G)$	$T(G,1)$
BARABASI (BAR)	Synthetic	365 488	734 347	68 500	6.41	5 198	18
BRIGHTKITE (BRI)	Social	33 187	188 577	5 680	1.75	2 326	18
CA-GRQC (CAG)	Collab.	2 651	10 480	499	0.03	1 271	18
CA-HEPTH (CAH)	Collab.	5 898	20 983	1 130	0.03	2 085	18
CAIDA (CAI)	Aut. Sys.	6 855	13 341	1 650	0.02	1 412	18
COM-YOUTUBE (CYT)	Social	452 060	2 295 072	66 200	5 090	4 136	18
DENMARK (DNK)	Road	252 416	320 914	147 000	0.75	13 152	18
FLICKREDGES (FED)	Social	105 512	2 316 450	2 180	165	12 415	18
FORESTFIRE (FF2)	Synthetic	1 178 888	13 849 776	12 500	16 100	17 983	18
FLICKRLINKS (FLI)	Social	704 985	14 501 930	125 000	17 700	12 525	18
OREGON (ORE)	Aut. Sys.	7 218	19 448	638	2.54	286	18
SKITTER (SKI)	Aut. Sys.	1 443 769	10 830 987	197 000	11 100	10 666	18
WIKIVOTE (WIK)	Hyperlinks	4 786	98 456	751	1.12	1 890	18
WIKITALK (WIT)	Collab.	622 315	2 889 703	47 700	38 700	2 951	18

TABLE 1. Input details and performance of PLL and KHL w.r.t. time and space, resp.

Then, for each network, we ran both PLL and Algorithm 4 (denoted by KHL in the following) and measured the computational time spent, in seconds, to build $P(G)$ and a 1-EFTPL $P(G,1)$, resp., which is shown in Table 1 (4th

[3] see networkit.iti.kit.edu

[4] see http://snap.stanford.edu

and 5th column, resp.). For the sake of completeness, we also measured the average size of $P(G)$ and the overhead required for storing $T(G,1)$, in bytes, per vertex. Such values are reported in Table 1 as well (6th and 7th column, resp.). Finally, we selected $10\,000$ pairs of vertices of the graph and, for each pair s,t, we executed the following steps:[5]

(1) Randomly removed an edge e of the graph which, with probability $p = 5/100$ is chosen to be on the shortest path between s and t in G;[6]
(2) Ran, on $G-e$, a modified version of Algorithm 3 that includes a *point-of-failure (POF) rerouting procedure*, with s and t as input, and measured both the time taken and the length of the output backup path; notice that, a POF rerouting procedure is a common backup method to deal with failures, that essentially retrieve a feasible solution starting from the point where a failure was encountered [53]; in our case, when one of the two sub-paths computed by Algorithm 3 happens to encounter a failed edge, say for instance the path from s fails at edge $e = (x,y)$, then we build a BFS in $G - e$ rooted at x and we stop as soon as t is reached;
(3) Ran Algorithm 5 on $G-e$ with s and t as input, and measured both the time taken and the length of the output backup path; iv) we compute the shortest-path between s and t on $G - e$ (e.g. by means of a BFS) and measure the length of the true shortest path between s and t in $G - e$. We accordingly compute, for pair s,t, the stretch of the path computed by ii) and that of the path computed by iii).

For all the considered performance metrics, we computed average values over the $10\,000$ executions. The results concerning the query time (in seconds) are reported in Fig. 2 while those related to the average stretch are shown in Fig. 1. We have omitted the stretch of the point-of-failure strategy, since it was always negligible, close to 1. Regarding the root vertex r of the $P(G,1)$, note that, from a worst-case point of view, the stretch provided by the approach is independent from the specific chosen vertex, hence it could be selected at random. However,

[5]All our software has been compiled with GNU g++ v.5 (O4 opt. level) under Linux and executed on an Intel Xeon© CPU
[6]the choice of p is inspired by previous works on failures in networks [38]

it is clear that the best choice of the root from a practical viewpoint should be based on some centrality measure, since this would tend to induce short backup paths [28]. Unfortunately, computing fine-grained centrality measures, such as, e.g. betweenness centrality, can be rather computationally expensive and therefore can (heavily) impact on the preprocessing time of our approach. Thus, when needed, we choose r to be the vertex of highest degree, since vertex degree has been shown to be a quite robust centrality measure that can be computed in linear time [23]. For the sake of completeness, we also considered an approximation of the betweenness centrality based on sampling, which however always produced worse results in terms of stretch, and thus we omitted them.

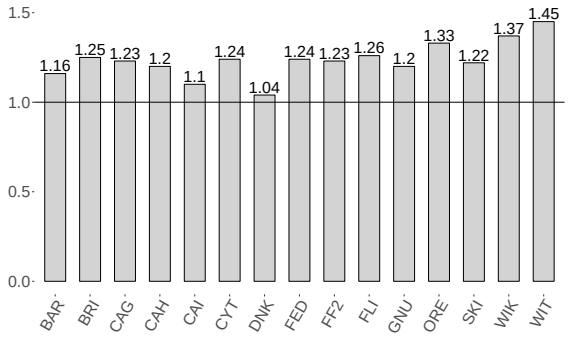

FIGURE 1. Estimation of $\alpha(P(G,1))$ based on $10\,000$ measures.

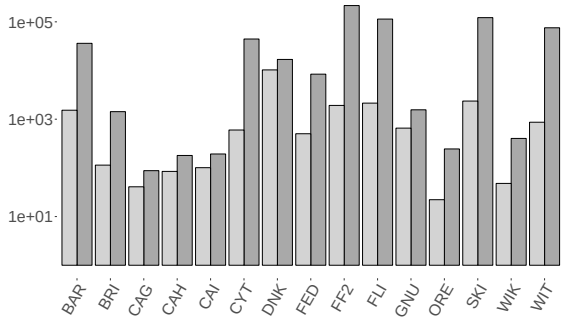

FIGURE 2. Query time of Algorithm 5 (lighter gray) versus query time of Algorithm 3 with POF rerouting.

7.3.1. Analysis. The main outcome of our experimental evaluation concerns both average stretch, provided by the new structure $P(G, 1)$, and query time. Regarding the former, despite the $O(kn)$ worst case upper bound of Lemma 7, the $P(G, 1)$ data structure shows a very good behavior in practice, as the measured average stretch is always pretty close to one, and only slightly larger than the stretch exhibited by the rerouting method, which can be considered a benchmark strategy w.r.t. quality of reported shortest paths. Regarding the latter, Algorithm 5 results in being extremely (orders of magnitude) faster (see Fig. 2) w.r.t. the rerouting-based strategy, which require to perform BFSs every time a failed edge is met. This suggests the method to be a viable option in practice whenever transient failures need to managed. This comes at the price of: i) a negligible time overhead for enriching the $P(G)$ data structure (i.e. for performing KHL), which can be hundreds of times smaller than the time for executing PLL; ii) a constant amount of extra space (few bytes per vertex) for storing $T(G, 1)$, which can be considered negligible w.r.t. the space per vertex required for storing $P(G)$.

CHAPTER 8

Conclusion

We have studied 2-*hop cover distance*/*path-reporting labeling schemes* in the fault-tolerant approximate setting. In particular, we have presented a simple and efficient way of enriching this successful scheme in order to make it *resilient* to the failure of a subset of k edges in the graph, by exploiting the notion of so-called *edge-independent* spanning trees.

Depending on k, we have provided different strategies and analyzed the corresponding performance. In addition, to assess the practical effectiveness of our method, we have provided an experimental evaluation, conducted for the case $k = 1$, whose results show the new method to be really competitive when compared with the performance of the benchmark non-fault-tolerant scheme, equipped with a point-of-failure rerouting policy.

There are several research directions which might deserve further investigation in this area. For instance, it could be interesting to study whether it is possible to design a scheme with a better guarantee on the stretch w.r.t. the one proposed in this thesis. Another interesting direction could be that of extending the experimental evaluation of our new method to higher values of k.

Bibliography

[1] Abraham, I., Bartal, Y., and Neiman, O. (2007). Embedding metrics into ultrametrics and graphs into spanning trees with constant average distortion. In *Proc. of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 502–511.

[2] Akiba, T., Iwata, Y., and Yoshida, Y. (2013). Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, pages 349–360. ACM.

[3] Alstrup, S., Holm, J., de Lichtenberg, K., and Thorup, M. (2005). Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264.

[4] Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., and Werneck, R. F. (2015). Route planning in transportation networks. *arXiv preprint arXiv:1504.05140*.

[5] Bilò, D., Colella, F., Gualà, L., Leucci, S., and Proietti, G. (2015a). A faster computation of all the best swap edges of a tree spanner. In *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015*, pages 239–253.

[6] Bilò, D., Colella, F., Gualà, L., Leucci, S., and Proietti, G. (2017). Effective edge-fault-tolerant single-source spanners via best (or good) swap edges. *arXiv preprint arXiv:1707.08861*.

[7] Bilò, D., Grandoni, F., Gualà, L., Leucci, S., and Proietti, G. (2015b). Improved purely additive fault-tolerant spanners. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 167–178.

[8] Bilò, D., Gualà, L., Leucci, S., and Proietti, G. (2014a). Fault-tolerant approximate shortest-path trees. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 137–148.

[9] Bilò, D., Gualà, L., Leucci, S., and Proietti, G. (2016a). Compact and fast sensitivity oracles for single-source distances. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, pages 13:1–13:14.

[10] Bilò, D., Gualà, L., Leucci, S., and Proietti, G. (2016b). Multiple-edge-fault-tolerant approximate shortest-path trees. In *Proc. of 33rd Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 47 of *LIPIcs*, pages 18:1–18:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[11] Bilò, D., Gualà, L., and Proietti, G. (2014b). Finding best swap edges minimizing the routing cost of a spanning tree. *Algorithmica*, 68(2):337–357.

[12] Bilò, D., Gualà, L., and Proietti, G. (2015c). A faster computation of all the best swap edges of a shortest paths tree. *Algorithmica*, 73(3):547–570.

[13] Brandstädt, A., Chepoi, V., and Dragan, F. F. (1999). Distance approximating trees for chordal and dually chordal graphs. *J. Algorithms*, 30(1):166–184.

[14] Brodal, G. S. and Jacob, R. (2002). Dynamic planar convex hull. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 617–626. IEEE.

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

[16] Chechik, S., Langberg, M., Peleg, D., and Roditty, L. (2009). Fault-tolerant spanners for general graphs. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 435–444.

[17] Cheriyan, J. and Maheshwari, S. (1988). Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs. *Journal of Algorithms*, 9(4):507–537.

[18] Cionini, A., D'Angelo, G., D'Emidio, M., Frigioni, D., Giannakopoulou, K., Paraskevopoulos, A., and Zaroliagis, C. D. (2014). Engineering graph-based models for dynamic timetable information systems. In *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, volume 42 of *OASICS*, pages 46–61. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[19] Cohen, E., Halperin, E., Kaplan, H., and Zwick, U. (2003). Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355.

[20] Curran, S., Lee, O., and Yu, X. (2006). Finding four independent trees. *SIAM J. Comput.*, 35(5):1023–1058.

[21] D'Andrea, A., D'Emidio, M., Frigioni, D., Leucci, S., and Proietti, G. (2015). Path-fault-tolerant approximate shortest-path trees. In *Proc. of 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 9439 of *Lecture Notes in Computer Science*. Springer.

[22] D'Angelo, G., D'Emidio, M., and Frigioni, D. (2014). A loop-free shortest-path routing algorithm for dynamic networks. *Theoretical Computer Science*, 516:1–19.

[23] D'Angelo, G., D'Emidio, M., and Frigioni, D. (2016). Distance queries in large-scale fully dynamic complex networks. In *Proc. of 27th International Workshop on Combinatorial Algorithms (IWOCA)*, volume 9843 of *Lecture Notes in Computer Science*, pages 109–121. Springer.

[24] D'Angelo, G., D'Emidio, M., Frigioni, D., and Maurizio, V. (2011). A speed-up technique for distributed shortest paths computation. In *Proc. of 11th International Conference on Computational Science and Its Applications (ICCSA)*, volume 6783 of *Lecture Notes in Computer Science*, pages 578–593. Springer.

[25] D'Angelo, G., D'Emidio, M., Frigioni, D., and Vitale, C. (2012). Fully dynamic maintenance of arc-flags in road networks. In Klasing, R., editor, *Proc. of 11th International Symposium on Experimental Algorithms (SEA)*, volume 7276 of *Lecture Notes in Computer Science*, pages 135–147. Springer.

[26] Das, S., Gfeller, B., and Widmayer, P. (2010). Computing all best swaps for minimum-stretch tree spanners. *J. Graph Algorithms Appl.*, 14(2):287–306.

[27] Datta, A. K., Larmore, L. L., Pagli, L., and Prencipe, G. (2013). Linear time distributed swap edge algorithms. In *Algorithms and Complexity, 8th International Conference, CIAC 2013, Barcelona, Spain, May 22-24, 2013. Proceedings*, pages 122–133.

[28] Delling, D., Goldberg, A. V., Pajor, T., and Werneck, R. F. (2014). Robust distance queries on massive networks. In *Proc. of 22th Annual European Symposium on Algorithms (ESA)*, volume 8737 of *Lecture Notes in Computer Science*, pages 321–333. Springer.

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

[30] Dinitz, M. and Krauthgamer, R. (2011). Fault-tolerant spanners: better and simpler. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 169–178.

[31] Emek, Y. and Peleg, D. (2008). Approximating minimum max-stretch spanning trees on unweighted graphs. *SIAM J. Comput.*, 38(5):1761–1781.

[32] Flocchini, P., Enriques, A. M., Pagli, L., Prencipe, G., and Santoro, N. (2004). Efficient protocols for computing the optimal swap edges of a shortest path tree. In *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*, pages 153–166.

[33] Flocchini, P., Enriques, A. M., Pagli, L., Prencipe, G., and Santoro, N. (2006). Point-of-failure shortest-path rerouting: Computing the optimal swap edges distributively. *IEICE Transactions*, 89-D(2):700–708.

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

[35] Gabow, H. N. (1985). A scaling algorithm for weighted matching on general graphs. In *26th Annual Symposium on Foundations of Computer Science (FOCS), Portland, Oregon, USA, 21-23 October 1985*, pages 90–100.

[36] Gualà, L. and Proietti, G. (2007). Exact and approximate truthful mechanisms for the shortest paths tree problem. *Algorithmica*, 49(3):171–191.

[37] Harel, D. and Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355.

[38] Iannaccone, G., Chuah, C.-n., Mortier, R., Bhattacharyya, S., and Diot, C. (2002). Analysis of link failures in an ip backbone. In *Proc. of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 237–242. ACM.

[39] Itai, A. and Rodeh, M. (1988). The multi-tree approach to reliability in distributed networks. *Inf. Comput.*, 79(1):43–59.

[40] Italiano, G. F. and Ramaswami, R. (1998). Maintaining spanning trees of small diameter. *Algorithmica*, 22(3):275–304.

[41] Ito, H., Iwama, K., Okabe, Y., and Yoshihiro, T. (2003). Polynomial-time computable backup tables for shortest-path routing. In *Structural Information and Communication Complexity - 10th International Colloquium, SIROCCO 2003, June 18-20, 2003, Umeå Sweden*, pages 163–177.

[42] Ito, H., Iwama, K., Okabe, Y., and Yoshihiro, T. (2005). Single backup table schemes for shortest-path routing. *Theor. Comput. Sci.*, 333(3):347–353.

[43] Jordan, C. (1869). Sur les assemblages de lignes. *J. Reine Angew. Math*, 70(185):81.

[44] Liebchen, C. and Wünsch, G. (2008). The zoo of tree spanner problems. *Discrete Applied Mathematics*, 156(5):569–587.

[45] Menger, K. (1927). Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115.

[46] Nardelli, E., Proietti, G., and Widmayer, P. (2001). A faster computation of the most vital edge of a shortest path. *Inf. Process. Lett.*, 79(2):81–85.

[47] Nardelli, E., Proietti, G., and Widmayer, P. (2003). Swapping a failing edge of a single source shortest paths tree is good and fast. *Algorithmica*, 35(1):56–74.

[48] Nardelli, E., Proietti, G., and Widmayer, P. (2004). Nearly linear time minimum spanning tree maintenance for transient node failures. *Algorithmica*, 40(2):119–132.

[49] Pettie, S. (2005). Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time. In *Algorithms and Computation, 16th International Symposium, ISAAC 2005, Sanya, Hainan, China, December 19-21, 2005, Proceedings*, pages 964–973.

[50] Proietti, G. (2000). Dynamic maintenance versus swapping: An experimental study on shortest paths trees. In *Algorithm Engineering, 4th International Workshop, WAE 2000, Saarbrücken, Germany, September 5-8, 2000*, pages 207–217.

[51] Qin, Y., Sheng, Q. Z., and Zhang, W. E. (2015). SIEF: efficiently answering distance queries for failure prone graphs. In *Proc. of 18th International Conference on Extending Database Technology (EDBT)*, pages 145–156. OpenProceedings.org.

[52] Roskind, J. and Tarjan, R. E. (1985). A note on finding minimum-cost edge-disjoint spanning trees. *Mathematics of Operations Research*, 10(4):701–708.

[53] Santoro, N. (2006). *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience.

[54] Tarjan, R. E. (1982). Sensitivity analysis of minimum spanning trees and shortest path trees. *Inf. Process. Lett.*, 14(1):30–33.

[55] Vieira, M. V., Fonseca, B. M., Damazio, R., Golgher, P. B., de Castro Reis, D., and Ribeiro-Neto, B. A. (2007). Efficient search ranking in social networks. In *Proc. of 16th ACM Conference on Information and Knowledge Management (CIKM)*, pages 563–572. ACM.

[56] Wu, B. Y., Hsiao, C., and Chao, K. (2008). The swap edges of a multiple-sources routing tree. *Algorithmica*, 50(3):299–311.

[57] Zhang, X. and Chen, L. (2017). Distance-aware selective online query processing over large distributed graphs. *Data Science and Engineering*, 2(1):2–21.