Noname manuscript No.
(will be inserted by the editor)

An Improved Algorithm for Computing All the Best Swap Edges of a Tree Spanner

Davide Bilò · Feliciano Colella · Luciano Gualà · Stefano Leucci · Guido Proietti

Received: date / Accepted: date

Abstract A *tree σ-spanner* of a positively real-weighted n-vertex and m-edge undirected graph G is a spanning tree T of G which approximately preserves (i.e., up to a multiplicative *stretch factor σ*) distances in G. Tree spanners with provably good stretch factors find applications in communication networks, distributed systems, and network design. However, finding an optimal or even a good tree spanner is a very hard computational task. Thus, if one has to face a *transient* edge failure in T, the overall effort that has to be afforded to rebuild a new tree spanner (i.e., computational costs, set-up of new links, updating of the routing tables, etc.) can be rather prohibitive. To circumvent this drawback, an effective alternative is that of associating with each tree edge a best possible (in terms of resulting stretch) *swap edge* – a well-established approach in the literature for several other tree topologies. Correspondingly, the problem of computing *all* the best swap edges of a tree spanner is a challenging algorithmic problem, since solving it efficiently means to exploit the structure of shortest paths not only in G, but also in all the scenarios in which an edge

D. Bilò
Università di Sassari, Italy
E-mail: davide.bilo@uniss.it

F. Colella
Gran Sasso Science Institute, L'Aquila, Italy
E-mail: feliciano.colella@gssi.it

L. Gualà
Università di Roma "Tor Vergata", Italy
E-mail: guala@mat.uniroma2.it

S. Leucci
ETH Zürich, Switzerland
E-mail: stefano.leucci@inf.ethz.ch

G. Proietti
Università degli Studi dell'Aquila, Italy
Istituto di Analisi dei Sistemi ed Informatica, CNR, Roma, Italy.
E-mail: guido.proietti@univaq.it

of T has failed. For this problem we provide a very efficient solution, running in $O(n^2 \log^4 n)$ time, which drastically improves (almost by a quadratic factor in n in dense graphs) on the previous known best result.

Keywords Transient edge failure · Swap algorithm · Tree spanner

1 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 $w : E(G) \to \mathbb{R}^+$, 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 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 [14, 20] 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. And 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. For instance, 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 [19], where α denotes the inverse of the Ackermann function, by using an efficient data structure, namely the *split-findmin* [12]. This was improving on another efficient solution given by Tarjan [22], 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* [6], *top trees* [3], *mergeable heaps* [18], and many others.

In this paper, we focus on the ABSE problem on the 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 distances in T and G, respectively. Correspondingly, an *optimal* tree spanner has minimum stretch out of all the spanning trees of G. Unfortunately, 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. Thus, swapping in a tree spanner is even more attractive than in general, and indeed the `ABSE-TS` problem was studied in [10], 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. Hence, in [3] 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 $O(m^2 \log \alpha(m,n))$ and $O(mn \log n)$ time, respectively, and both using linear space. Notice that from a computational point of view, as shown in [3], the two settings are substantially equivalent, so our solutions can be used to improve the results given in [10] as well.

1.1 Our result

In this paper, 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 the running time of both the algorithms provided in [3], 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.

To put into focus our result, it is worth noticing that, as observed in [10], 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. And 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:

- 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.
- 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.

1.2 Related work

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 [9]), then the problem is not approximable within any constant factor better than 2, unless $\mathsf{P}=\mathsf{NP}$ [16]. 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 $\mathsf{P}=\mathsf{NP}$, the problem is not approximable within an additive term of $o(n)$ [11]. 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 [9], 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 [7] and the references therein).

Concerning the problem of swapping in 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), the *minimum routing-cost spanning tree* (MRCST), and the *single-source shortest-path tree* (SPT). Concerning the MDST, a best swap is instead an edge minimizing the *diameter* of the swap tree [13,17], and the best solution runs in $O(m \log \alpha(m,n))$ time [6]. Regarding the MRCST, a best swap is clearly an edge minimizing the *all-to-all routing cost* of the swap tree [23], and the fastest solution for solving this problem has a running time of $O\left(m 2^{O(\alpha(n,n))} \log^2 n\right)$ [5]. Concerning the SPT, the most prominent swap criteria are those aiming to minimize either the maximum or the average distance from the root, and the corresponding ABSE problems can be addressed in $O(m \log \alpha(m,n))$ time [6] and $O(m\, \alpha(n,n) \log^2 n)$ time [21], respectively. Recently, in [4], the authors proposed two new criteria for swapping in a SPT, which are in a sense related with this paper, namely the minimization of the maximum and the average stretch factor from the root, for which they proposed an efficient $O(mn + n^2 \log n)$ and $O(mn \log \alpha(m,n))$ time solution, respectively.

Finally, 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].

1.3 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 a spanning tree T of G, and 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. After providing a high-level description of our approach, we will explain in detail how it works, by organizing our analysis as specified in the next section.

2 High-level description of the algorithm

Let us consider the tree T spanning G as rooted at any fixed arbitrary vertex. W.l.o.g., and for the sake of simplifying the exposition of our algorithm, we can assume that T is binary. Indeed, if this is not the case, then we can transform G and T into an equivalent graph G' with weight function w', 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 works as follows.

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.

Clearly, $|V(G')| = O(|V(G)|)$, $|E(G')| = O(|E(G)|)$, and, 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 iff 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 iff f is a BSE for the edge (z, v_i) w.r.t. T', where z is the parent of v_i in T'.

Thus, let us assume T is binary. 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 $n/2$ [15]. 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

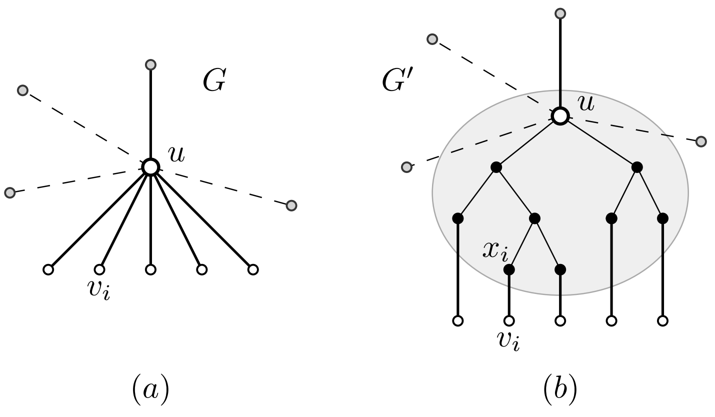

Fig. 1 Reducing the degree of vertices in G: on the left side, the tree T (solid edges) embedded in G, on the right side the superimposition of the binary tree to T in order to get a maximum degree of 3. Solid edges in the gray area have weight 0, while the weight of (x_i, v_i) is $w(u, v_i)$.

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 Algorithm `ABSE-TS`) works in $n-1$ phases, one for each tree edge as considered in preorder w.r.t. T, and at the end of each phase returns a BSE for that edge. 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) for e, i.e., an edge $f_v \in \arg\min_{f \in S(e,v)} \sigma_{G-e}(T_{e/f})$, and (ii) the corresponding stretch factor $\sigma_{G-e}(T_{e/f_v})$. To simplify handling of special cases, whenever $S(e, v) = \emptyset$, we assume that $f_v = \perp$ and that $\sigma_{G-e}(T_{e/f_v}) = +\infty$. Once all the v-BSEs for e are computed, a BSE f^* for e can be found as the one minimizing the associated stretch factor.

The core of our algorithm is exactly the efficient computation of a v-BSE and of its stretch factor. As we will discuss in more detail in Section 3, this is done through a clever selection of a set of $O(\log n)$ *candidate* v-BSEs, each of which is evaluated against $O(\log^2 n)$ *candidate* critical edges (see Section 4). As we will show in Section 5, each of these latter candidates can be efficiently selected in $O(\log n)$ time, by dynamically maintaining the *upper envelopes* of a set of linear functions expressing the criticality of an edge w.r.t. a given candidate swap edge. In this way, a total of $O(\log^3 n)$ pairs swap-critical edge

Algorithm: ABSE-TS(G, T)

```
1 $\mathcal{T} \leftarrow$ Centroid decomposition of $T$;
2 foreach $e \in E(T)$ in postorder do                                   // $n-1$ phases
3     $U_e \leftarrow$ vertices of the component of $T - e$ that contains the root of $T$;
4     $f^* \leftarrow \perp$;                                            // Current BSE for $e$
5     foreach $v \in U_e$ do                                             // $O(n)$ sub-phases
6         compute a $v$-BSE $f_v$ for $e$ and $\sigma_{G-e}(T_{e/f_v})$;  // This takes $O(\log^4 n)$ time
7         if $\sigma_{G-e}(T_{e/f_v}) < \sigma_{G-e}(T_{e/f^*})$ then $f^* \leftarrow f_v$;
8     return $f^*$ as BSE for $e$ and continue with the next phase.
```

are evaluated, at a cost of $O(\log n)$ time each, and thus in $O(\log^4 n)$ we are able to retrieve a v-BSE for the currently considered tree edge e.

3 Selecting a candidate best swap edge

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 [3]:

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 $O(\log n)$.

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)):

[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.

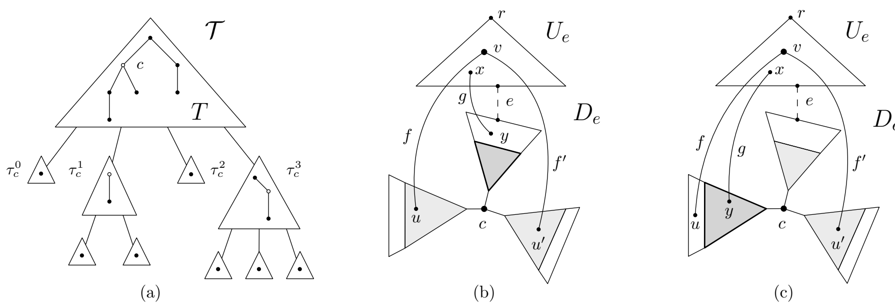

Fig. 2 (a) An example of centroid decomposition of the tree T (which corresponds to the first vertex of $\mathcal{T}$). (b) and (c): Two of the four possible cases situation illustrated in Lemma 1. The subtree $\widehat{T}$ is represented by the three gray triangles along with the vertex c. f is a swap 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 $\widehat{T}$.

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:

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

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` (notice that v and e are fixed). 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. Procedure `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.*

Procedure FindBCE(Λ)

```
1  if $|V(\Lambda)| = 0$ then return $(\bot, \bot)$ ;
2  $c \leftarrow$ Centroid of $\Lambda$;
3  if $c \in U_e$ then
4      $\tau \leftarrow$ unique child of $\Lambda$ in $\mathcal{T}$ that contains all the vertices in $V(\Lambda) \cap D_e$;
5      return FindBCE $(\tau)$;
6  else                                                            // $c \in D_e$
7      Compute an edge $f = (v, u) \in \arg\min_{(v,u) \in S(e,v)} \{w(v, u) + d_T(u, c)\}$;
8      $g_1 = (x, y) \leftarrow$ FindCritical$(f, T)$;    // Compute a critical edge for $f$ (see
        Sec. 4)
9      $\tau \leftarrow (c, y)$-tree of $\Lambda$;       // Either $\tau$ is empty or it is a child of $\Lambda$ in $\mathcal{T}$
10     $(f', g_2) \leftarrow$ FindBCE $(\tau)$;
11     if $\phi(f, g_1) \leq \phi(f', g_2)$ then return $(f, g_1)$; else return $(f', g_2)$;
```

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.

Next lemma provides an upper bound to the running time of the procedure:

Lemma 3 *Procedure* ***FindBCE***(T) *requires* $O((\Gamma_{\texttt{cand}} + \Gamma_{\texttt{FC}}) \log n)$ *time, where* $\Gamma_{\texttt{cand}}$ *and* $\Gamma_{\texttt{FC}}$ *is the time required to perform Steps 7 and 8, i.e., the time to find a candidate v-BCE 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, Procedure `FindCritical` will require $O(\log^3 n)$ time and $O(m \log^2 n)$ space, as we will show in the rest of the paper. On the other hand, we now show that $\Gamma_{\texttt{cand}} = O(\log n)$, after a preprocessing time and space of $O(n^2)$, by making use of *top-trees* [2]. 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

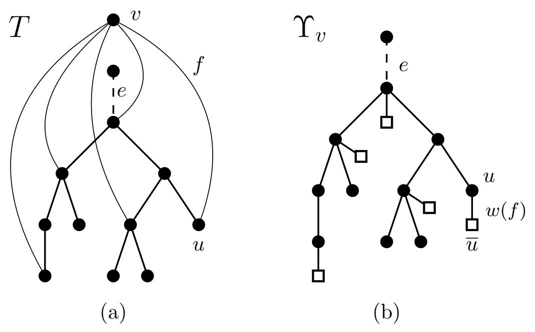

Fig. 3 (a) The subtree of T induced by D_e along with swap edges in $S(e,v)$, and (b) the corresponding top-tree Υ_v. The black vertices of Υ_v are the same of the tree T. For each $u \in V(T)$ such that $f = (v,u) \in S(e,v)$, Υ_v contains an additional vertex $\overline{u}$ (shown as a white square),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

can be *marked* and the top-tree is able to perform *closest marked vertex* (CMV, for short) queries, i.e., it can report the 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).

We can then give the following:

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, and their space usage is $O(n^2)$.*

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 $d_{T'}(c,\overline{u}) = d_{T'}(c,u) + d_{T'}(u,\overline{u}) = d_T(c,u) + w(f)$.

4 Selecting a candidate critical edge

To implement Procedure `FindCritical` in the promised $O(\log^3 n)$ time, we will compute $O(\log^2 n)$ candidate critical edges for f, by paying $O(\log n)$ time to select each one of them, and as anticipated one out of them will be a critical edge for f.

More precisely, we look at $O(\log n)$ subtrees of the centroid decomposition $\mathcal{T}$ and, for each such subtree $\mathit{\Lambda}$, we will consider $O(\log n)$ subtrees $\mathit{\Psi}$ to find a critical edge candidate having one endpoint in $\mathit{\Psi}$ and the other in $\mathit{\Lambda}$. The choice of the $O(\log^2 n)$ pairs of trees is guided by the position of f, while the computation of a candidate for a given pair $(\mathit{\Psi}, \mathit{\Lambda})$ is the core of the procedure, and is performed efficiently through the dynamic maintenance of the upper envelope of a set of linear functions, as described in the next section.

Definition 4 ($(\mathit{\Psi}, \mathit{\Lambda})$-Critical Edge) Given a failing edge e and a swap edge $f = (v, u) \in S(e, v)$, and given two trees $\mathit{\Psi}, \mathit{\Lambda}$ of the centroid decomposition $\mathcal{T}$, a $(\mathit{\Psi}, \mathit{\Lambda})$-*critical edge* for f is an edge

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

When $\mathit{\Psi} = T$ we will refer to a $(\mathit{\Psi}, \mathit{\Lambda})$-critical edge as a $\mathit{\Lambda}$-critical edge.

Let $f = (v, u) \in S(e, v)$ and let $\mathit{\Lambda}$ be a tree of the centroid decomposition $\mathcal{T}$ such that $u \in V(\mathit{\Lambda})$. Procedure `FindCritical` returns a $\mathit{\Lambda}$-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 $V(\mathit{\Lambda}) \cap D_e$). 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` uses Procedure `FindCriticalCandidate`$(f, \mathit{\Psi}, \mathit{\Lambda})$ as a subroutine, which for the sake of clarity will be described in the next subsection. For the moment, it suffices to know that `FindCriticalCandidate` receives three inputs, i.e., edge $f = (v, u)$ and two subtrees $\mathit{\Psi}, \mathit{\Lambda}$ of the centroid decomposition $\mathcal{T}$ such that $v \in \mathit{\Psi}$ and, either $u \notin V(\mathit{\Lambda})$ or $\mathit{\Lambda}$ is the tree containing the sole vertex u, and it returns a $(\mathit{\Psi}, \mathit{\Lambda})$-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 $\mathit{\Lambda}$ be a tree of the centroid decomposition $\mathcal{T}$ such that $u \in V(\mathit{\Lambda})$. Procedure `FindCritical`$(f, \mathit{\Lambda})$ returns a $\mathit{\Lambda}$-critical edge for f.*

Proof The proof is by induction on the cardinality of $V(\mathit{\Lambda})$.

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

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

```
1 if $V(\Lambda) = \{u\}$ then return FindCriticalCandidate($f$, $T$, $\Lambda$) ;
2 $c \leftarrow$ Centroid of $\Lambda$;
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($f, \tau_c^j$) ;
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($f, \tau_c^j$);
8 return $\arg\max_{g \in \{g_1, g_2\}} \{\phi(f, g)\}$;
```

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 containing one or more vertices in D_e (as otherwise the vertices in $V(\Lambda) \cap D_e$ would be 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 induction hypothesis, that it returns a τ_c^j-critical edge for f, which is also Λ-critical edge for f.

Lemma 6 *Procedure* ***FindCritical***(f, Λ) *requires* $O(\Gamma_{\mathtt{FCC}} \log n)$ *time, where* $\Gamma_{\mathtt{FCC}}$ *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 $\Gamma_{\mathtt{FCC}} = O(\log^2 n)$.

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

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

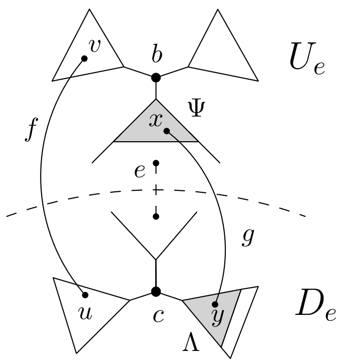

Fig. 4 Illustration of Lemma 8. f is a swap edge for e, Ψ and Λ are two trees of the centroid decomposition, and b and c are their corresponding parent centroids. g is a potential (Ψ, Λ)-critical edge for f. Notice that the unique path from x to v (resp. from y to u) passes through b (resp. c).

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 it 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 complementary case 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 that were the case, then the 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) traverses 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 Procedure `FindCriticalCandidate`). 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, Procedure `FindCriticalCandidate`(f, Ψ, Λ) computes a (Ψ, Λ)-critical edge for f.*

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

```
1 if $V(\Lambda) \cap D_e = \emptyset$ then return $\perp$;
2 if $V(\Psi) = \{v\}$ then return $\mathcal{Q}_e(f, \Psi, \Lambda)$;
3 $b \leftarrow$ Centroid of $\Psi$;
4 Let $j$ be the unique index in $\{0, 1, 2, 3\}$ such that $v \in V(\tau_b^j)$;
5 if $b \in D_e$ then return FindCriticalCandidate($f, \tau_b^j, \Lambda$);
6 $\mathcal{G} \leftarrow \{\mathcal{Q}_e(f, \tau_b^i, \Lambda) : i = 0, 1, 2, 3 \wedge i \neq j\}$;     // Here $b \in U_e$
7 $g_1 \leftarrow \arg\max_{g \in \mathcal{G}} \phi(f, g)$;
8 $g_2 \leftarrow$ FindCriticalCandidate($f, \tau_b^j, \Lambda$);
9 return $\arg\max_{g \in \{g_1, g_2\}} \{\phi(f, g)\}$;
```

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 $|V(\Psi)|$. 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 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 there is at most one τ_b^i containing at least one vertex in U_e (as the converse 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 we know, by induction hypothesis, that it returns a τ_b^j-critical edge for f, which is also (Ψ, Λ)-critical edge for f.

Lemma 10 *Procedure* `FindCriticalCandidate`*(f, Ψ, Λ) requires $O(\Gamma_{\mathcal{Q}_e} \log n)$ time, 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 we explain in details in the next section.

5 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. Here 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 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 [8].

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. Then, we have the following:

Lemma 11 *The query $\mathcal{Q}_e(f, \Psi, \Lambda)$ (used in `FindCriticalCandidate`) can be executed in $O(\log n)$ amortized time.*

Proof 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.

5.1 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.

5.2 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.

The merge operation consists of two steps: the *join step* and the *update step.*

5.2.1 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 $\nu_{e_1} + \nu_{e_2}$.

5.2.2 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.

5.3 Analysis

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

Lemma 12 *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 13 *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 14 *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 12 and Lemma 13 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.

From the above lemmas, we can now prove the following:

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 12 and Lemma 13 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 14 we have that every function is moved $O(\log n)$ times. Since there are $O(m \log^2 n)$ functions, as shown by Lemma 12, and a function can be moved in $O(\log n)$ time, we have that the total time spent moving functions is $O(m \log^4 n)$.

By combining Lemmas 3, 4, 6, 10, 11 and Proposition 2, we can finally give our main result:

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.*

6 Conclusion

In this paper we have provided a new time-efficient algorithm for finding all the best swap edges of a tree spanner. This has been obtained by suitably combining a centroid decomposition of the tree spanner along with an efficient dynamic maintenance of the upper envelopes of a set of linear functions. We believe that our approach may be useful to solve other related swap problems.

Although our improvement was substantial as compared to the state of the art, the problem of designing an $o(n^2)$ time algorithm remains a challenging open problem, even for the unweighted case. Another interesting research direction is that of studying the swap problem on the *minimum average-stretch* tree spanner, for which a solution whose average stretch is $O(1)$ away from the distances in the underlying graph can be computed in polynomial time [1]. Finally, we mention the related problem of computing *good* swap edges for a tree spanner, namely sub-optimal swap edges that can be computed fast (ideally, in linear time), and whose induced stretch factor is provably close to that provided by a corresponding best swap edge.

References

1. Abraham, I., Bartal, Y., Neiman, O.: 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, pp. 502–511 (2007)
2. Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M.: Maintaining information in fully dynamic trees with top trees. ACM Transactions on Algorithms **1**(2), 243–264 (2005). DOI 10.1145/1103963.1103966
3. Bilò, D., Colella, F., Gualà, L., Leucci, S., Proietti, G.: A faster computation of all the best swap edges of a tree spanner. In: Proc. of the 22nd Intl. Colloquium Structural Information and Communication Complexity, pp. 239–253 (2015). DOI 10.1007/978-3-319-25258-2_17
4. Bilò, D., Colella, F., Gualà, L., Leucci, S., Proietti, G.: Effective edge-fault-tolerant single-source spanners via best (or good) swap edges. In: Proc. of the 24th Intl. Colloquium Structural Information and Communication Complexity (in press)
5. Bilò, D., Gualà, L., Proietti, G.: Finding best swap edges minimizing the routing cost of a spanning tree. Algorithmica **68**(2), 337–357 (2014). DOI 10.1007/s00453-012-9674-y
6. Bilò, D., Gualà, L., Proietti, G.: A faster computation of all the best swap edges of a shortest paths tree. Algorithmica **73**(3), 547–570 (2015). DOI 10.1007/s00453-014-9912-6
7. Brandstädt, A., Chepoi, V., Dragan, F.F.: Distance approximating trees for chordal and dually chordal graphs. J. Algorithms **30**(1), 166–184 (1999). DOI 10.1006/jagm.1998.0962
8. Brodal, G.S., Jacob, R.: Dynamic planar convex hull. In: Proc. of the 43rd Annual IEEE Symposium on Foundations of Computer Science, pp. 617–626. IEEE (2002)
9. Cai, L., Corneil, D.G.: Tree spanners. SIAM J. Discrete Math. **8**(3), 359–387 (1995). DOI 10.1137/S0895480192237403
10. Das, S., Gfeller, B., Widmayer, P.: Computing all best swaps for minimum-stretch tree spanners. J. Graph Algorithms Appl. **14**(2), 287–306 (2010)
11. Emek, Y., Peleg, D.: Approximating minimum max-stretch spanning trees on unweighted graphs. SIAM J. Comput. **38**(5), 1761–1781 (2008). DOI 10.1137/060666202
12. Gabow, H.N.: A scaling algorithm for weighted matching on general graphs. In: Proc. of the 26th Annual Symposium on Foundations of Computer Science, pp. 90–100 (1985). DOI 10.1109/SFCS.1985.3
13. Italiano, G.F., Ramaswami, R.: Maintaining spanning trees of small diameter. Algorithmica **22**(3), 275–304 (1998). DOI 10.1007/PL00009225
14. Ito, H., Iwama, K., Okabe, Y., Yoshihiro, T.: Polynomial-time computable backup tables for shortest-path routing. In: Proc. of the 10th Intl. Colloquium Structural Information and Communication Complexity, pp. 163–177 (2003)
15. Jordan, C.: Sur les assemblages de lignes. J. Reine Angew. Math **70**(185), 81 (1869)
16. Liebchen, C., Wünsch, G.: The zoo of tree spanner problems. Discrete Applied Mathematics **156**(5), 569–587 (2008). DOI 10.1016/j.dam.2007.07.001
17. Nardelli, E., Proietti, G., Widmayer, P.: A faster computation of the most vital edge of a shortest path. Inf. Process. Lett. **79**(2), 81–85 (2001). DOI 10.1016/S0020-0190(00)00175-7
18. Nardelli, E., Proietti, G., Widmayer, P.: Nearly linear time minimum spanning tree maintenance for transient node failures. Algorithmica **40**(2), 119–132 (2004). DOI 10.1007/s00453-004-1099-9
19. Pettie, S.: Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time. In: Proc. of the 16th Intl. Symposium on Algorithms and Computation, pp. 964–973 (2005). DOI 10.1007/11602613_96
20. Proietti, G.: Dynamic maintenance versus swapping: An experimental study on shortest paths trees. In: Proc. of the 4th Intl. Workshop on Algorithm Engineering, pp. 207–217 (2000). DOI 10.1007/3-540-44691-5_18
21. Salvo, A.D., Proietti, G.: Swapping a failing edge of a shortest paths tree by minimizing the average stretch factor. Theor. Comput. Sci. **383**(1), 23–33 (2007). DOI 10.1016/j.tcs.2007.03.046
22. Tarjan, R.E.: Sensitivity analysis of minimum spanning trees and shortest path trees. Inf. Process. Lett. **14**(1), 30–33 (1982). DOI 10.1016/0020-0190(82)90137-5
23. Wu, B.Y., Hsiao, C., Chao, K.: The swap edges of a multiple-sources routing tree. Algorithmica **50**(3), 299–311 (2008). DOI 10.1007/s00453-007-9080-z