Algorithmica (2015) 73:547–570
DOI 10.1007/s00453-014-9912-6

A Faster Computation of All the Best Swap Edges of a Shortest Paths Tree

Davide Bilò · Luciano Gualà · Guido Proietti

Received: 27 December 2013 / Accepted: 17 June 2014 / Published online: 12 July 2014

Abstract We consider a two-edge connected, non-negatively real-weighted graph G with n vertices and m edges, and a single-source shortest paths tree (SPT) of G rooted at an arbitrary vertex. If an edge of the SPT is temporarily removed, a widely recognized approach to reconnect the vertices disconnected from the root consists of joining the two resulting subtrees by means of a single non-tree edge, called a *swap edge*. This allows to reduce consistently the set-up and computational costs which are incurred if one instead rebuilds a new optimal SPT from scratch. In the past, several optimality criteria have been considered to select a *best* possible swap edge, and here we restrict our attention to arguably the two most significant measures: the minimization of either the *maximum* or the *average* distance between the root and the disconnected vertices. For the former criteria, we present an $O(m \log \alpha(m, n))$ time algorithm—where α is the inverse of the Ackermann function—to find a best swap edge for *every* edge of the SPT, thus improving onto the previous $O(m \log n)$ time algorithm. Concerning the latter criteria, we provide an $O(m + n \log n)$ time algorithm for the special but important case where G is *unweighted*, which compares

D. Bilò
Dipartimento di Scienze Umanistiche e Sociali, University of Sassari, Sassari, Italy
e-mail: davide.bilo@uniss.it

L. Gualà
Dipartimento di Ingegneria dell'Impresa, University of Rome "Tor Vergata", Rome, Italy
e-mail: guala@mat.uniroma2.it

G. Proietti (✉)
Dip. di Ingegneria e Scienze dell'Informazione e Matematica, Univ. of L'Aquila, L'Aquila, Italy
e-mail: proietti@di.univaq.it

G. Proietti
Istituto di Analisi dei Sistemi ed Informatica, CNR, Rome, Italy

favourably with the $O\left(m + n\,\alpha(n,n)\log^2 n\right)$ time bound that one would get by using the fastest algorithm known for the weighted case—once this is suitably adapted to the unweighted case.

Keywords Single-source shortest paths tree · Edge fault tolerance · Swap algorithms

1 Introduction

In communication networking, *broadcasting* a message from a source node to every other node of the network is one of the most common operations. Since this should be done by making use of a logical communication topology as sparser and faster as possible, then it is quite natural to resort to a single-source shortest paths tree (SPT) rooted at the source node. However, despite its popularity, the SPT is highly susceptible to link malfunctioning, as any tree-based network topology: the smaller the number of links is, the higher the average traffic for each link is, and the bigger the risk of a link overloading is. Even worse, the failure of a single link may cause the disconnection of a large part of the network.

In principle, two different approaches can be adopted to solve the problem of a link failure: either we rebuild a new SPT from scratch (which can be very expensive in terms of computational and set-up costs), or we quickly reconnect the two subtrees induced by the link failure by *swapping* it with a single non-tree edge (see [12,17] for some practical motivations supporting this latter approach). Quite obviously, swapping requires that the *swap edge* is wisely selected so that the resulting *swap tree* is as much efficient as possible in terms of some given post-swap distance measure from the root to the just reconnected nodes. Moreover, to be prepared to any possible failure event, it makes sense to study the problem of dealing with the failure of *each and every* single link in the network. This defines a so-called all-best-swap edges (ABSE) problem on the SPT [3,13].

1.1 Related Work

The problem of swapping in spanning trees has received a significant attention from the algorithmic community. There is indeed a line of papers which address ABSE problems starting from different types of spanning trees. Just to mention a few, we recall here the minimum spanning tree (MST), the minimum diameter spanning tree (MDST), and the minimum routing-cost spanning tree (MRCST) of a given two-edge connected, non-negatively real-weighted graph G having n vertices and m edges. For the MST, a best swap is of course a swap edge minimizing the *weight* of the swap tree, i.e., a swap edge of minimum weight.[1] This problem is also known as the MST *sensitivity analysis* problem. Concerning the MDST, a best swap is instead an edge minimizing the *diameter* of the swap tree [11,14]. Finally, for the MRCST, a best swap is clearly an edge minimizing the *all-to-all routing cost* of the swap tree [19].

[1] Notice that for the MST, edge weights can also be negative, and the swap tree is actually an MST of the graph deprived of the failed edge.

The fastest solutions for solving the corresponding ABSE problems have a running time of $O(m \log \alpha(m, n))$ [16], $O(m \log n)$ [8], and $O\left(m 2^{O(\alpha(2m,2m))} \log^2 n\right)$ [2], respectively, where α is the inverse of the Ackermann function originally defined in [1].

Getting back to the SPT, the appropriate definition of a best swap seems however more ambiguous, since an SPT is actually the *union* of a set of shortest paths emanating from a root vertex, and so there is not a univocal global optimization measure we have to aim at when swapping. Thus, in [15], where the corresponding ABSE problem was initially studied, several different criteria expressing desirable features of the swap tree of an SPT were introduced in order to characterize a best possible swap edge. In particular, among all of them, two can be viewed as the most prominent ones: the *maximum* and the *average* (or, equivalently, the *total*) distance from the root to the disconnected vertices. These two measures reflect a classic egalitarian *versus* utilitarian viewpoint as far as the efficiency of the swap is concerned. From the algorithmic side, the fastest known solutions for the two problems amount to $O(m \log n)$ (as a by-product of the result in [8]) and to $O(m \alpha(m, m) \log^2 n)$ time (see [6][2]), respectively.

It is worth noticing that swapping in an SPT can be reviewed as fast and good at the same time: in fact, recomputing every new optimal SPT from scratch would require as much as $O(mn \log \alpha(m, n))$ time [9] (no faster dynamic algorithm is indeed known). Moreover, it has been shown that in the swap tree as computed w.r.t. the maximum (resp., average) criteria, the maximum (resp., average) distance of the disconnected vertices from the root is at most twice (resp., three times) that of the new optimal SPT [15].

1.2 Our Results

In this paper, we focus exactly on the ABSE problem on an SPT w.r.t. the maximum and average distance measures. To this respect, for the former criteria we present an $O(m \log \alpha(m, n))$ time algorithm. As we will see, our result generalizes to the ABSE problem on an MDST, and thus, we improve the time complexity of $O(m \log n)$ given in [8] for both problems. It is worth noticing that our approach beats the $O(m \log n)$ time barrier needed to sort the edges w.r.t. their weight, and we match the time complexity of the best-known algorithm for the MST sensitivity analysis problem on real-weighted graphs. In our opinion, this is particularly remarkable since the MST sensitivity analysis problem can be reduced in linear time to our problem.[3] Thus, improving the time complexity of our algorithm would provide a faster algorithm for performing a sensitivity analysis of an MST, which is one of the main open problems in the area of MST related computations.

[2] Actually, in [6] the authors claim an $O(m \log^2 n)$ time bound, but this must be augmented by an $O(\alpha(m, m))$ factor, as pointed out in [2].

[3] Indeed, we can perform a sensitivity analysis of an MST as follows: (i) we first make the graph G non-negatively weighted, by summing up to all the edge weights the absolute value of the lightest edge, then (ii) we root the MST at any arbitrary vertex, hence (iii) we set the weight of every edge of the MST to 0, and finally (iv) we solve the ABSE problem w.r.t. the maximum distance from the root. Clearly, for every edge e of the MST, the best swap edge computed by the algorithm is an edge of minimum weight cycling with e.

As far as the second criteria is concerned, we focus on the special but important case where G is *unweighted*. In this case we are indeed able to first exhibit a *sparsification* technique which reduces to $O(n)$ the number of non-tree edges and thus immediately allows to use the $O(m\,\alpha(m,m)\log^2 n)$ time algorithm known for the weighted case (see [6]) so as to obtain a faster $O\left(m + n\,\alpha(n,n)\log^2 n\right)$ time solution. However, we go beyond this improvement by designing an even more efficient $O(m + n\log n)$ time algorithm which runs on this sparser graph and uses sophisticated data structures. Unfortunately, the extension of this algorithm to weighted graphs sounds hard, and so we regard this as a challenging open problem. It is worth noticing that the sparsification technique can be successfully applied also to the first criteria. Therefore, thanks to our algorithm for weighted graphs, the ABSE problem on an SPT w.r.t. the maximum distance measure can be solved in $O(m+n\log\alpha(n,n))$ for the class of unweighted graphs.

The paper is organized as follows: in Sect. 2 we describe a preprocessing step which will be used to guarantee the efficiency of our algorithms, while in Sect. 3 and 4, we present our algorithms for the maximum and the average distance criteria, respectively. Finally, Sect. 5 contains the conclusions.

2 A Preprocessing Step

In this section we present our notation and describe a useful preprocessing step which allows us to simplify the solution of the ABSE problems we address in this paper.

Let $G = (V(G), E(G), w)$ be a non-negatively real-weighted, undirected, and two-edge-connected graph, with n vertices and m edges, where $V(G)$ denotes the set of vertices, $E(G)$ denotes the set of edges, and w denotes the edge-weight function. Let T be an SPT of G rooted at an arbitrary vertex r; thus, T contains a shortest path from r to every other vertex of G. In the rest of the paper, we will call an edge of T a *tree edge* and an edge in $E(G)\setminus E(T)$ a *non-tree edge*. For a vertex $v \neq r$, we denote by $\hat{v}$ the *parent* of v in T (in particular, if $\hat{v} \neq r$, then $\hat{\hat{v}}$ denotes the parent of $\hat{v}$). We denote by $\texttt{lca}(v, v')$, the *least common ancestor* of the vertices $v, v' \in V(T)$, i.e., the vertex of T farthest from r that is an ancestor of both v and v'. Let $x \neq r$ be a vertex of G and let $e = (\hat{x}, x)$ be the tree edge between x and its parent $\hat{x}$. We denote by T_x the subtree of T rooted at x and containing all the descendants in T of x. We denote by $C_e = \{(u, y) \in E(G)\setminus E(T) : (u \in V(G)\setminus V(T_x)) \wedge (y \in V(T_x))\}$ the set of *swap edges* for e, i.e., the non-tree edges that may be used to reconnect the two subtrees induced by the removal of e from T (see Fig. 1).

Since G is two-edge-connected, we have that $C_e \neq \emptyset$ for every tree edge e. Let in the following $T_{e/f}$ denote the *swap tree* obtained from T by swapping e with $f \in C_e$. For any two vertices $v, v' \in V(T)$, we finally denote by $d(v, v')$ and $d_{e/f}(v, v')$ the distance between v and v' in T and $T_{e/f}$, respectively.

Depending on the goal that we pursue in swapping, some swap edge may be preferable to some other one. Here we focus on the following problems:

1. max-ABSE: for every $e = (\hat{x}, x) \in E(T)$, find an edge $f_e \in C_e$ s.t.:

$$f_e \in \arg\min_{f\in C_e}\left\{\max_{v\in V(T_x)} d_{e/f}(r, v)\right\};$$

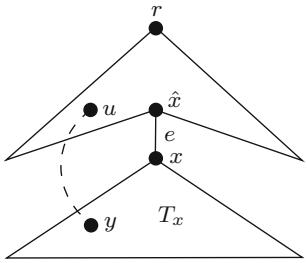

Fig. 1 This figure shows some of the notation used in the paper. The tree T is represented by *solid lines* and polygons while a swap edge (u, y) of e is *dashed*

2. `sum`-ABSE: for every $e = (\hat{x}, x) \in E(T)$, find an edge $f_e \in C_e$ s.t.:

$$f_e \in \arg\min_{f \in C_e} \left\{ \sum_{v \in V(T_x)} d_{e/f}(r, v) \right\}.$$

Notice that, by definition, f_e minimizes the *average* distance from r to the disconnected vertices.

To solve efficiently our ABSE problems, we first transform in a standard way (see for instance [18]) each non-tree edge $f = (u, y)$ into two *vertical* non-tree edges, i.e., $f' = (\texttt{lca}(u, y), y)$ and $f'' = (\texttt{lca}(u, y), u)$, each of them with an appropriate weight, namely $w(f') = d(r, u) + w(f) - d(r, \texttt{lca}(u, y))$ and $w(f'') = d(r, y) + w(f) - d(r, \texttt{lca}(u, y))$, respectively. Basically, $w(f')$ (resp., $w(f'')$) once added to $d(r, \texttt{lca}(u, y))$, is the length of the path in $T_{e/f}$ starting from r, passing through f, and ending in y (resp., u), after that f has swapped with an edge e along the path from y (resp., u) to $\texttt{lca}(u, y)$. Next, as long as there are pairwise parallel vertical non-tree edges, we keep the cheapest of them (ties can be broken arbitrarily) and discard all the others. In this way, we obtain an auxiliary (multi)graph G' with at most twice the number of non-tree edges of G and which is perfectly equivalent to G as far as the study of our ABSE problems is concerned. Notice that this transformation can be performed for all non-tree edges in $O(m)$ time, once that the SPT is given, since essentially it only requires the computation of the least common ancestors of non-tree edges and of the distances from the root to every vertex [10]. In the rest of the paper, we will therefore assume to be working on G', unless differently stated, and that for a swap edge $f = (u, y)$ of G', vertex $y \in V(T_x)$, and so $u = \texttt{lca}(u, y)$.

3 A Faster Algorithm for `max`-ABSE

In this section we provide the description of an algorithm solving the `max`-ABSE problem in $O(m \log \alpha(m, n))$ time and $O(m)$ space. As we will see, our result generalizes to the ABSE problem on an MDST. Thus, for both problems we improve the time complexity of $O(m \log n)$ given in [8]. For the sake of simplifying the exposition, we assume that edge weights are strictly positive in the rest of this section. However, our algorithm can be easily adapted for the class of non-negatively real-weighted graphs, by defining suitable tie-breaking rules.

3.1 Further Notation and Basic Properties

We need to introduce some further notation and prove some basic properties that will be used for the algorithm description and correctness, respectively. We denote by $P(v, v')$ the (unique) path in T between two vertices v and v', and by $w\left(P(v, v')\right) := d(v, v')$ the *length* of $P(v, v')$. A *diametral path* of T is a longest path in T. A *center* of T is a vertex c of T that minimizes the length of the longest path in T starting at it, i.e., $\max_{v \in V(T)} d(c, v) \leq \max_{v \in V(T)} d(v', v)$, for every $v' \in V(T)$. It is a folklore result that a positively real-weighted tree has either one center or two centers and, if a tree has two centers, then they are adjacent. In the rest of the paper, w.l.o.g., for every $v \in V(T)$ and every $v' \in V(T_v)$, we will assume that the longest path in T_v starting at v' is unique.[4] This implies that the diametral path of every subtree T_v of T (T included) is unique. Let v be an inner vertex of T and let P be the longest path starting at a center c of T_v and entirely contained in T_v. We call the edge of P incident to c the *midpoint edge of* T_v and we denote by c_v (resp., c'_v) the end vertex of the midpoint edge of T_v which is farthest from (resp., closest to) r. If v is a leaf, then $c_v, c'_v := v$. It is worth noticing that the definition of midpoint edge of T_v is independent of the chosen tree center. Indeed, if T_v has two centers, then the midpoint edge of T_v is the edge between the two centers of T_v.

The following lemma—which was already proved in [8]—shows a useful property satisfied by longest paths and midpoint edges.

Lemma 1 ([8]) *Let $e = (\hat{x}, x)$ be a tree edge such that x is an inner vertex of T, and let q_x and q'_x be the two endpoints of the diametral path P_x of T_x, respectively. For every $v \in V(T_x)$, either $P = P(v, q_x)$ or $P = P(v, q'_x)$ is the longest path in T_x starting at v. Furthermore the midpoint edge of T_x is contained in P.*

Let $f = (u, y)$ be a swap edge of $e = (\hat{x}, x)$ and let q_x and q'_x be the endvertices of the diametral path P_x of T_x. W.l.o.g. let q_x be a descendant of c_x (so, q'_x is not a descendent of c_x) (see Fig. 2). Lemma 1 implies that

$$\begin{aligned}\max_{v \in V(T_x)} d_{e/f}(r, v) &= d(r, u) + w(f) + \begin{cases} d(y, q_x) & \text{if} \quad \texttt{lca}(y, c_x) \neq c_x, \\ d(y, q'_x) & \text{otherwise,} \end{cases} \\ &= d(r, u) + w(f) + d(r, y) + \begin{cases} d(r, q_x) - 2d(r, \texttt{lca}(y, q_x)) & \text{if} \quad \texttt{lca}(y, c_x) \neq c_x, \\ d(r, q'_x) - 2d(r, \texttt{lca}(y, q'_x)) & \text{otherwise.} \end{cases} \end{aligned} \tag{1}$$

Equation (1) shows that the objective function value for every tree edge e and every swap edge f of e can be computed in constant time once we know both the distances in T from all vertices to r, and the diametral path P_x of T_x for all the internal vertices x of T. Furthermore, as we will see, it allows to compare efficiently (some) pairs of

[4] Indeed, we can arbitrarily number the tree vertices with different numbers, and then choose, among all paths of equal length starting at v', the one minimizing the number associated with its endpoint different from v'.

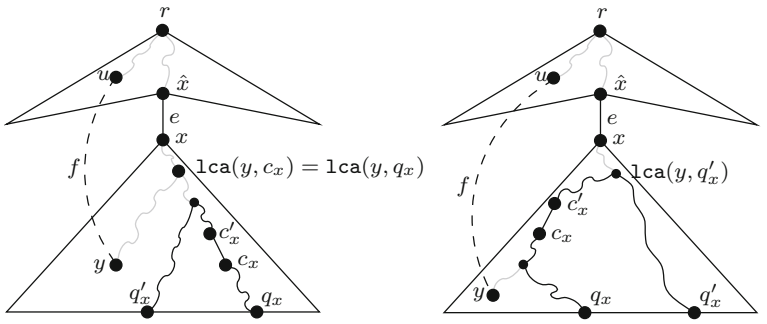

Fig. 2 This figure shows the tree structure when edge $e = (\hat{x}, x)$ is swapped with the *dashed* non-tree edge $f = (u, y)$. The diametral path of T_x is depicted with a *black curved line* that highlights the midpoint edge of T_x. All other paths in T are represented with *gray curved lines*. The picture on the *left-hand side* covers the case $\texttt{lca}(y, c_x) \neq c_x$ while the picture on the *right-hand side* covers the other case (i.e., $\texttt{lca}(y, c_x) = c_x$)

swap edges $f = (u, y)$ and $f' = (u', y')$ for an edge $e = (\hat{x}, x)$ on the basis of the relationship between $\texttt{lca}(y, c_x)$ and $\texttt{lca}(y', c_x)$, because of the many common terms they will share in Eq. (1).

Next lemma provides a useful structural property satisfied by the vertices c_v's which is also used by our algorithm.

Lemma 2 *Let v and v' be two distinct vertices of T such that v' is an ancestor of v in T. Then either $c_{v'}$ is a vertex of $P(v, c_v)$ or $c_{v'} \in V(T) \setminus V(T_v)$.*

Proof Let P_v and $P_{v'}$ be the diametral paths in T_v and $T_{v'}$, respectively. We prove the claim by showing that if $c_{v'} \in V(T_v)$, then $c_{v'}$ is a vertex of $P(v, c_v)$. Thus, we assume that $c_{v'} \in V(T_v)$. By definition, c_v (resp., $c_{v'}$) is a vertex of P_v (resp., $P_{v'}$). Notice that if $P_{v'}$ is entirely contained in T_v, then trivially $P_v = P_{v'}$ and the claim follows. Otherwise, we have $v \in V(P_{v'})$, as $c_{v'} \in V(T_v)$. Since $P_{v'}$ is the diametral path of $T_{v'}$ and T_v is a subtree of $T_{v'}$, the maximal (w.r.t. vertex addition) subpath of $P_{v'}$ contained in T_v, say P, is the longest path of T_v starting at v. From Lemma 1, P contains the midpoint edge of T_v. Therefore, $P_{v'}$ also contains the midpoint edge of T_v. As $w(P_{v'}) \geq w(P_v)$, we have that $c_{v'}$ must be an ancestor of c_v, and the claim follows. □

3.2 High-Level Description of the Algorithm

Our algorithm traverses the tree edges in a suitable preorder and, for each tree edge $e = (\hat{x}, x)$, it computes a best swap edge f_e of e in four steps with a clever implementation of the approach used in [8]. More precisely, for each tree edge $e = (\hat{x}, x)$, our algorithm does the following:

Step 1: for every vertex $v \in V(T_x)$, it computes a best possible swap edge of e among the set of swap edges of e which are incident to v, say $f(v)$ (if no such edge exists, then, with a little abuse of notation, we assume that $f(v)$ is an imaginary edge of weight $+\infty$);

Step 2: it partitions $V(T_x)$ into $|V(P(x, c_x))|$ *groups*, where each vertex z of $P(x, c_x)$ defines the group $\mathcal{G}(x, z) := \{v \in V(T_x) \mid \texttt{lca}(v, c_x) = z\}$;[5]

Step 3: for every vertex z of $P(x, c_x)$, it computes a *group candidate*, i.e., a best possible swap edge of e among the edges in $F_z = \{f(v) \mid v \in \mathcal{G}(x, z)\}$, say $f(F_z)$;

Step 4: it selects a best swap edge f_e of e among the group candidates computed in Step 3.

Let $f = (u, y)$ be a swap edge of e and let $\texttt{lca}(y, c_x) = z$. Observe that z is a vertex of $P(x, c_x)$. Let $\kappa_1(f) := d(r, u) + w(f) + d(r, y)$ be the *primary key* associated with f, and let $\kappa_2(f, e) := \kappa_1(f) - 2d(r, z)$ be the *secondary key* associated with the pair f and e. The primary key of f is independent of the failing tree edge and is used to compare two competing swap edges in F_z, as shown in the next lemma which was already proved (in a slightly different way, though) in [8].

Lemma 3 ([8]) *Let $e = (\hat{x}, x)$ be a tree edge and let $f = (u, y)$, $f' = (u', y')$ be two swap edges of e such that $\texttt{lca}(y, c_x) = \texttt{lca}(y', c_x)$. If $\kappa_1(f) \leq \kappa_1(f')$, then $\max_{v \in V(T_x)} d_{e/f}(r, v) \leq \max_{v \in V(T_x)} d_{e/f'}(r, v)$.*

Proof As $\texttt{lca}(y, c_x) = \texttt{lca}(y', c_x)$, Eq. (1) implies that

$$\begin{aligned}
&\max_{v \in V(T_x)} d_{e/f}(r, v) - \max_{v \in V(T_x)} d_{e/f'}(r, v) \\
&= d(r, u) + w(f) + d(r, y) - (d(r, u') + w(f') + d(r, y')) \\
&= \kappa_1(f) - \kappa_1(f').
\end{aligned}$$

Therefore, $\kappa_1(f) \leq \kappa_1(f')$ iff $\max_{v \in V(T_x)} d_{e/f}(r, v) \leq \max_{v \in V(T_x)} d_{e/f'}(r, v)$. The claim follows. □

The secondary key, which depends on z and thus on the failing tree edge e, is used to compare two competing swap edges among the group candidates $f(F_z)$'s with $z \neq c_x$, as shown in the next lemma.

Lemma 4 *Let $e = (\hat{x}, x)$ be a tree edge and let $f = (u, y)$, $f' = (u', y')$ be two swap edges such that $\texttt{lca}(y, c_x), \texttt{lca}(y', c_x) \neq c_x$. If $\kappa_2(f, e) \leq \kappa_2(f', e)$, then $\max_{v \in V(T_x)} d_{e/f}(r, v) \leq \max_{v \in V(T_x)} d_{e/f'}(r, v)$.*

[5] Observe that, if $z = c_x$, then $\mathcal{G}(x, z) = V(T_{c_x})$; if $z \neq c_x$, then $\mathcal{G}(x, z) = V(T_z) \setminus V(T_v)$, where v is the (unique) child of z in $P(x, c_x)$.

Proof Let q_x be the (unique) endpoint of the diametral path of T_x which is a descendent of c_x. If $\kappa_2(f, e) \leq \kappa_2(f', e)$, then using Eq. (1), we have that

$$\begin{aligned}
\max_{v \in V(T_x)} d_{e/f}(r, v) &= d(r, u) + w(f) + d(r, y) + d(r, q_x) - 2d(r, \mathtt{lca}(y, c_x)) \\
&= \kappa_1(f) - 2d(r, \mathtt{lca}(y, c_x)) + d(r, q_x) \\
&= \kappa_2(f, e) + d(r, q_x) \\
&\leq \kappa_2(f', e) + d(r, q_x) \\
&= \kappa_1(f) - 2d(r, \mathtt{lca}(y', c_x)) + d(r, q_x) \\
&= d(r, u') + w(f') + d(r, y') + d(r, q_x) - 2d(r, \mathtt{lca}(y', c_x)) \\
&= \max_{v \in V(T_x)} d_{e/f'}(r, v).
\end{aligned}$$

This concludes the proof. □

3.3 Algorithm Implementation

Next three corollaries of Lemmas 3 and 4 provide useful properties suitable for an efficient implementation of Step 1, Step 3, and Step 4 of our algorithm, respectively.

Corollary 1 *Let $e = (\hat{x}, x)$ be a tree edge, let v be a vertex of T_x, and, if $\hat{x} \neq r$, let $f'(v)$ be a best possible swap edge of $e' = (\hat{\hat{x}}, \hat{x})$ among the swap edges of e' which are incident to v. We have that*

$$f(v) = \begin{cases} (\hat{x}, v) & \text{if} \quad (\hat{x}, v) \in C_e \text{ and } \left(\hat{x} = r \text{ or } \kappa_1((\hat{x}, v)) < \kappa_1(f'(v))\right), \\ f'(v) & \text{otherwise.} \end{cases}$$

Proof Let

$$X = \begin{cases} \emptyset & \text{if} \quad \hat{x} = r, \\ \{(u, y) \in C_{e'} \mid y = v\} & \text{otherwise.} \end{cases}$$

Observe that

$$\{(u, y) \in C_e \mid y = v\} = \begin{cases} X \cup \{(\hat{x}, v)\}. & \text{if} \quad (\hat{x}, v) \in C_e, \\ X & \text{otherwise.} \end{cases}$$

Since the primary key is independent of the failing edge, $f'(v)$ is a best possible swap edge of e among the non-tree edges in X. Therefore, using Lemma 3, $f(v) = f'(v)$ unless $(\hat{x}, v) \in C_e$ and either $\hat{x} = r$ or $\kappa_1((\hat{x}, v)) < \kappa_1(f'(v))$. The claim follows. □

Corollary 2 *Let $e = (\hat{x}, x)$ be a tree edge and let z be a vertex of $P(x, c_x)$. We have that $f(F_z) \in \arg\min\{\kappa_1(f) \mid f \in F_z\}$.*

Proof The proof is a direct consequence of Lemma 3 as, by definition, for every edge $f = (u, y) \in F_z$, $\texttt{lca}(y, c_x) = z$. □

Corollary 3 *Let $e = (\hat{x}, x)$ be a tree edge and let $f(F_x, \ldots, F_{c'_x}) \in \arg\min\{\kappa_2(f(F_z), e) \mid z \in V(P(x, c'_x))\}$. Then, one of the edges $f(F_{c_x})$ and $f(F_x, \ldots, F_{c'_x})$ is a best swap edge of e.*

Proof By definition, $f(F_z)$ is a best possible swap edge of e among the swap edges which are selected in F_z. Therefore, as a consequence of Lemma 4, $f(F_x, \ldots, F_{c'_x})$ is a best possible swap edge of e among the edges in the set $X = \bigcup_{z \in V(P(x,c'_x))} F_z$. The claim now follows because the set $F_{c_x} \cup X$ contains all the best possible swap edges for each $v \in V(T(x))$. □

The following corollary is an immediate consequence of Lemma 2 and shows that, thanks to a suitable preorder processing of the tree edges, we can compute the new groups $\mathcal{G}(x, z)$, and thus implement Step 2 of the algorithm, by suitably splitting some groups $\mathcal{G}(\hat{x}, z')$ (that the algorithm already computed in previous steps).

Corollary 4 *Let $x \neq r$ be a vertex of T and let*

$$v = \begin{cases} c_{\hat{x}} & \text{if } x \text{ is a vertex of } P(\hat{x}, c_{\hat{x}}), \\ \hat{x} & \text{otherwise.} \end{cases}$$

We have that

(a) $\mathcal{G}(x, z) = \mathcal{G}(\hat{x}, z)$ *for every* $z \in V(P(x, c_x)) \cap V(P(\hat{x}, c'_{\hat{x}}))$;

(b) $\mathcal{G}(x, c_x) = \mathcal{G}(\hat{x}, c_{\hat{x}})$ *iff* $c_{\hat{x}} = c_x$;

(c) $\mathcal{G}(x, z) \subseteq \mathcal{G}(\hat{x}, v)$ *for every* $z \in V(P(x, c_x)) \setminus V(P(\hat{x}, c'_{\hat{x}}))$.

Therefore, to compute groups efficiently and, at the same time, to store information about $f(v)$'s and $f(F_z)$'s and the values of their primary and secondary keys, respectively, our algorithm uses *split-findmin* data structures which are suitable for finding minimum-key elements of sets that can only be split. A split-findmin is a data structure that has been successfully used in [16] to solve the sensitivity analysis of an MST in $O(m \log \alpha(m, n))$ time. A split-findmin $\mathcal{S}$ maintains a set of sequences of elements, where a key is associated with each element, and supports the following operations:

`init`$(s_1, \ldots, s_N)$: initializes the sequence $\mathcal{S} := \{(s_1, \ldots, s_N)\}$ of N elements with key $\kappa(s_i) := +\infty$ for all i;
`split`$(\mathcal{S}, s_i)$: let $s = (s_j, \ldots, s_{i-1}, s_i, \ldots, s_k)$, with $j < i \leq k$, be the sequence in $\mathcal{S}$ containing s_i; the call of `split`$(\mathcal{S}, s_i)$ returns $\mathcal{S} := (\mathcal{S} \setminus \{s\}) \cup \{(s_j, \ldots, s_{i-1}), (s_i, \ldots, s_k)\}$;
`findmin`$(\mathcal{S}, s_i)$: let s be the sequence in $\mathcal{S}$ containing s_i; the call of `findmin` $(\mathcal{S}, s_i)$ returns an element of minimum key in s;
`decreasekey`$(\mathcal{S}, s_i, k)$: if $\kappa(s_i) > k$, then `decreasekey`$(\mathcal{S}, s_i, k)$ sets $\kappa(s_i) = k$.

As proved in [16], a split-findmin of N elements requires $\Theta(N)$ space, can be initialized in $\Theta(N)$ time, and supports M split, findmin, and decreasekey operations in time $O(M \log \alpha(M, N))$.

Our algorithm uses two split-findmins $\mathcal{S}_1$ and $\mathcal{S}_2$ both containing all the vertices of T as elements. During the visit of the tree edge $e = (\hat{x}, x)$, the split-findmin $\mathcal{S}_1$ is updated so as:

($\mathcal{I}_1$): the value of the key associated with each $v \in V(T_x)$ in $\mathcal{S}_1$ is equal to $\kappa_1(f(v))$;
($\mathcal{I}_2$): for every vertex z of $P(x, c_x)$, $\mathcal{G}(x, z)$ is a sequence in $\mathcal{S}_1$.

Thus, because of ($\mathcal{I}_1$) and ($\mathcal{I}_2$) and thanks to Corollary 2, we can compute $f(F_z)$ via a `findmin`$(\mathcal{S}_1, z)$ query for every vertex z in $P(x, c_x)$. The split-findmin $\mathcal{S}_2$ is instead used to represent the group candidates $f(F_z)$ of all the groups $\mathcal{G}(x, z)$, with $z \in V(P(x, c'_x))$. More precisely, during the visit of the tree edge $e = (\hat{x}, x)$, $\mathcal{S}_2$ is updated so as the following invariants are maintained:

($\mathcal{I}_3$): for every $v \in V(T_x)$, the value of the key associated with v in $\mathcal{S}_2$ is equal to (a) $\kappa_2(f(v), e)$, if v is a vertex of $P(x, c'_x)$, or (b) $+\infty$, otherwise;
($\mathcal{I}_4$): $V(T_x)$ is a sequence in $\mathcal{S}_2$.

Because of ($\mathcal{I}_3$) and ($\mathcal{I}_4$) and thanks to Corollary 3, we can compute $f(F_x, \ldots, F_{c'_x})$ by performing a `findmin`$(\mathcal{S}_2, x)$ query. To maintain invariants ($\mathcal{I}_2$) and ($\mathcal{I}_4$), our algorithm does the following. First, it arranges T in such a way that, for each non-leaf vertex v, the subtree rooted at the leftmost child of v contains c_v. Observe that, because of Lemma 2, after such an arrangement c_v is a vertex of the path starting at v and ending at the leftmost leaf of T_v. Then, it initializes $\mathcal{S}_1$ and $\mathcal{S}_2$ with the sequence of vertices as obtained from a right-to-left preorder visit of T. Finally, it visits all the vertices of T in left-to-right preorder.

During the visit of vertex x, the algorithm "removes" one after the other a set of edges of T forming a subpath of the path in T between x and the midpoint edge of x, as explained in more details later. Actually, the algorithm does not really remove edges from T, but it rather simulates tree edge removals by means of splits in $\mathcal{S}_1$ and $\mathcal{S}_2$, as remarked by the following

Observation 1 *Let T' be a tree rooted at r', let s be the sequence of vertices as obtained from a right-to-left preorder visit of T', let $\mathcal{S}$ be a split-findmin containing the sequence s, and let v be the leftmost child of r'. The execution of* `split`$(\mathcal{S}, v)$ *splits s into two sequences s' and s'' such that, w.l.o.g., s' contains the vertices in $V(T'_v)$ arranged in a right-to-left preorder visit of T'_v, and s'' contains the vertices in $V(T') \setminus V(T'_v)$ arranged in a right-to-left preorder visit of the corresponding subtree of T'.*[6]

Let x be the vertex that is visited by the algorithm and, if $x \neq r$, let $e = (\hat{x}, x)$ be the corresponding tree edge. The algorithm follows the four-step approach described above and suitable updates both $\mathcal{S}_1$ and $\mathcal{S}_2$ to compute a best swap edge f_e of e.

[6] Notice that, from a topological point of view, the `split`$(\mathcal{S}, v)$ operation can be viewed as the removal of edge (r', v) from T'.

Furthermore, our algorithm is designed so as invariant $(\mathcal{I}_j)$ is maintained at the end of the execution of Step j.

For the base case $x = r$, the algorithm performs only Step 2, i.e., it calls $\texttt{split}(\mathcal{S}_1, z)$ for every $z \in V(P(r, c_r)) \setminus \{r\}$, to initialize $\mathcal{S}_1$ properly, thus maintaining invariant $(\mathcal{I}_2)$ (observe that the other invariants are already maintained as no swap edge needs to be computed and $T_r = T$).

Step 1 of the algorithm is implemented by calling $\texttt{decreasekey}(\mathcal{S}_1, v, \kappa_1(f))$ for every vertex $v \in V(T_x)$ such that $f = (\hat{x}, v)$ is a non-tree edge, and thus a swap edge of e. Indeed, if $\hat{x} = r$, then $f(v) = f$ from Corollary 1. Otherwise, if $x \neq r$, then, from Corollary 1, maintaining invariant $(\mathcal{I}_1)$ requires the comparison of $\kappa_1(f)$ with the value of the primary key of a best possible swap edge of $(\hat{\hat{x}}, \hat{x})$ incident to v which, because of the preorder processing of the tree vertices, is the value of the key associated with vertex v in $\mathcal{S}_1$.

Step 2 of the algorithm is implemented by calling $\texttt{split}(\mathcal{S}_1, z)$ for every $z \in V(P(x, c_x)) \setminus V(P(\hat{x}, c_{\hat{x}}))$. Indeed, because of the arrangement of the tree vertices and the left-to-right preorder processing of the tree vertices, at the beginning of Step 2, while visiting x, a sequence of $\mathcal{S}_1$ corresponds to a subtree of T, say T', such that the path from x to the leftmost leaf of T' contains the subpath induced by the vertices in $V(P(x, c_x)) \setminus V(P(\hat{x}, c_{\hat{x}}))$. Therefore, using Corollary 4 and repeated applications of Observation 1, we can show that invariant $(\mathcal{I}_2)$ is maintained at the end of Step 2 (see also Fig. 3).

Step 3 of the algorithm consists of the following operations which are performed for each group, say $\mathcal{G}(x, z)$, whose corresponding sequence in $\mathcal{S}_1$ was created or modified in the previous steps during the visit of x.[7] The algorithm first calls $\texttt{findmin}(\mathcal{S}_1, z)$, which, thanks to Corollary 2 and invariants $(\mathcal{I}_1)$ and $(\mathcal{I}_2)$, returns the group candidate $f(F_z)$, and, if $z \neq c_x$, it calls $\texttt{decreasekey}(\mathcal{S}_2, z, \kappa_2(f(F_z), e))$ to update the set of group candidates in $\mathcal{S}_2$ so as invariant $(\mathcal{I}_3)$ is maintained at the end of Step 3. Observe that if the sequence corresponding to a group $\mathcal{G}(x, z)$ is not modified in Step 1 or Step 2 while the algorithm visits x, then $\hat{x} \neq r$, $\mathcal{G}(\hat{x}, z) = \mathcal{G}(x, z)$, and the set of swap edges of e incident to vertices in $\mathcal{G}(x, z)$ is equal to the set of swap edges of $(\hat{\hat{x}}, \hat{x})$ incident to vertices of $\mathcal{G}(\hat{x}, z)$, i.e., $\{(u, y) \in C_e \mid y \in \mathcal{G}(x, z)\} = \{(u, y) \in C_{(\hat{\hat{x}}, \hat{x})} \mid y \in \mathcal{G}(\hat{x}, z)\}$. Therefore, from Corollarys 1 and 2, a best possible swap edge of $(\hat{\hat{x}}, \hat{x})$ among the edges in $\{(u, y) \in C_{(\hat{\hat{x}}, \hat{x})} \mid y \in \mathcal{G}(\hat{x}, z)\}$ is also a best possible swap edge of e among the edges in $\{(u, y) \in C_e \mid y \in \mathcal{G}(x, z)\}$.

Step 4 of the algorithm consists of (a) computing $f(F_x, \ldots, F_{c'_x})$ (the explanation of how to do it is few lines below), (b) calling $\texttt{findmin}(\mathcal{S}_1, c_x)$ which, thanks to Corollary 2 and invariants $(\mathcal{I}_1)$ and $(\mathcal{I}_2)$, returns the group candidate $f(F_{c_x})$, and (c) selecting a best swap edge of e between $f(F_x, \ldots, F_{c'_x})$ and $f(F_{c_x})$ in constant time using Eq. (1). To compute $f(F_x, \ldots, F_{c'_x})$, the algorithm first performs $\texttt{split}(\mathcal{S}_2, x)$ which—thanks to the right-to-left preorder arrangement of the tree vertices in $\mathcal{S}_2$ and to the left-to-right preorder processing of the tree edges—maintains the invariant $(\mathcal{I}_4)$ because of Observation 1, and then calls $\texttt{findmin}(\mathcal{S}_2, x)$.

[7] Observe that a decreasekey operation on $\mathcal{S}_1$ modifies one group while a split operation on $\mathcal{S}_1$ splits one group into two groups, i.e., it creates two new groups.

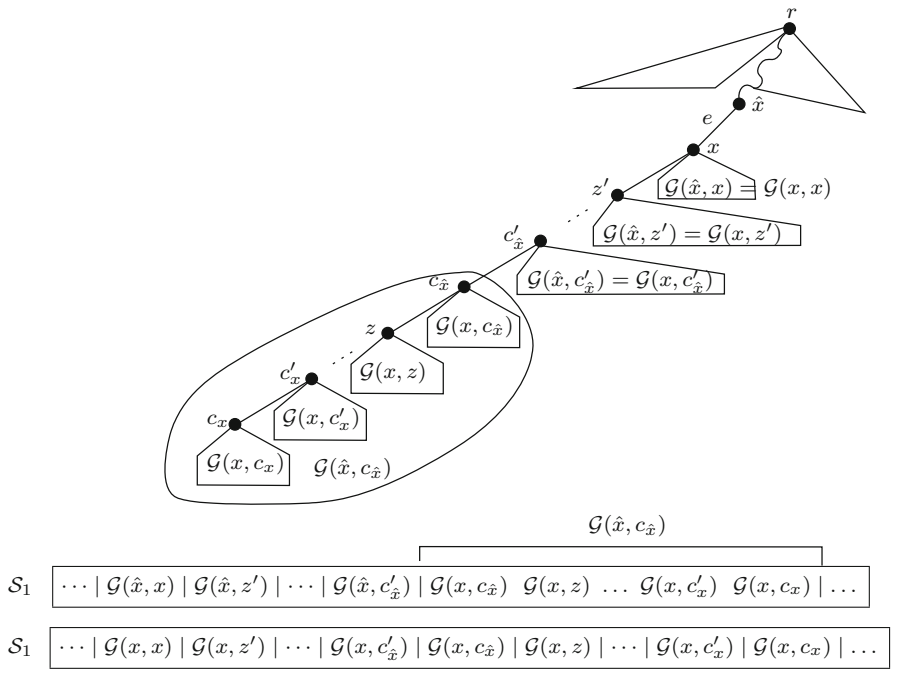

Fig. 3 An example showing the computation of the groups $\mathcal{G}(x, z)$ for the case in which x is the leftmost child of $\hat{x}$. Notice that $c_{\hat{x}}$ is an ancestor of c_x. The configurations of the split-findmin $\mathcal{S}_1$ before and after the algorithm visits x are both represented graphically one on top of the other. More precisely, of the two drawings, the topmost one is the configuration of $\mathcal{S}_1$ right before the algorithm visits x while the bottommost one is the configuration of $\mathcal{S}_1$ at the end of Step 2 while the algorithm visits x

The pseudocode of the algorithm is given in Algorithm 1. Algorithm 1 uses two vectors S and S' of n elements each. According to our notation, when tree edge $e = (\hat{x}, x)$ is considered, then $S[v]$ stores edge $f(v)$, while $S'[z]$ stores edge $f(F_z)$. Finally, it also uses a set L_x that contains a vertex v for each group $\mathcal{G}(x, v)$ that is created or modified in the first two steps of the visit of x.

Next theorem analyses the time and space complexity of Algorithm 1.

Theorem 1 *Algorithm 1 solves the* max-ABSE *problem in* $O(m \log \alpha(m, n))$ *time and* $O(m)$ *space.*

Proof The correctness of the algorithm can be easily shown using induction w.r.t. the order in which the tree vertices are visited and by proving that invariant $(\mathcal{I}_j)$ is maintained at the end of Step j, for every $j = 1, 2, 3, 4$. About its time and space complexity, first of all, we recall that the computation of the multigraph G' requires linear time and we also recall that G' has n vertices and at most $2m$ edges (see Sect. 2). Furthermore, for every vertex v, we can compute the midpoint edge of T_v and the endvertices of the diametral path of T_v in linear time by using a simple dynamic programming algorithm that processes the tree vertices in a bottom-up fashion. Furthermore, thanks to the preorder labelling of the tree vertices, the computation of the

Algorithm 1: Algorithm for max-ABSE.

Input : a two-edge-connected, non-negatively real-weighted graph $G = (V(G), E(G), w)$, a root vertex $r \in V(G)$, and an SPT T of G rooted at r.
Output: a best swap edge f_e for every $e \in E(T)$ w.r.t. max-ABSE.

Preprocess G so as explained in Section 2 to obtain a multigraph G' that contains T and non-parallel vertical non-tree edges;
for *every vertex v w.r.t. any fixed postorder visit of the tree vertices* **do**
  compute the midpoint edge (c'_v, c_v) of T_v and the endvertices q_v and q'_v of the diametral path of T_v so as q_v is a descendant of c_v;
  arrange T so as c_v is in the subtree rooted at the leftmost child of v;
  $E_v = \{(u, y) \in E(G') \setminus E(T) \mid v = \texttt{lca}(u, y)\}$
let s be the sequence of vertices of the right-to-left preorder visit of T;
$\mathcal{S}_1 = \texttt{init}(s)$; $\mathcal{S}_2 = \texttt{init}(s)$; $S[v]$, $S'[v] = \perp$, for every $v \in V(T)$;
for *every $z \in V(P(r, c_r)) \setminus \{r\}$* **do**
  $\texttt{split}(\mathcal{S}_1, z)$; **/* STEP 2 for r */**
for *every vertex $x \neq r$ w.r.t. the left-to-right preorder visit of T* **do**
  /* COMPUTE A BEST SWAP EDGE OF $(\hat{x}, x)$ */
  $L_x = \emptyset$;
  /* STEP 1 */
  for *every $f = (\hat{x}, y) \in E_{\hat{x}}$* **do**
    if $S[y] = \perp$ *or* $\kappa_1(f) < \kappa_1(S[y])$ **then**
      $\texttt{decreasekey}(\mathcal{S}_1, y, \kappa_1(f))$; $S[y] = f$; if $\texttt{lca}(y, c_x) \neq c_x$, add it to L_x;
  /* STEP 2 */
  for *every $z \in V(P(x, c_x)) \setminus V(P(\hat{x}, c_{\hat{x}}))$* **do**
    $\texttt{split}(\mathcal{S}_1, z)$;
  for *every $z \in V(P(x, c'_x)) \setminus V(P(\hat{x}, c_{\hat{x}}))$* **do**
    add z to L_x;
  /* STEP 3 */
  for *every $z \in L_x$* **do**
    $v = \texttt{findmin}(\mathcal{S}_1, z)$;
    if $\big(S'[z] = \perp$ *or* $\kappa_2(S[v]) < \kappa_2(S'[z])\big)$ **then**
      $\texttt{decreasekey}(\mathcal{S}_2, z, \kappa_2(S[v]))$; $S'[v] = S[v']$;
  /* STEP 4 */
  $\texttt{split}(\mathcal{S}_2, x)$;
  $v' = \texttt{findmin}(\mathcal{S}_2, x)$; $v = \texttt{findmin}(\mathcal{S}_1, c_x)$;
  if $S'[v'] = \perp$ **then**
    $f_{(\hat{x},x)} = S[v]$;
  else
    if $S[v] = \perp$ **then**
      $f_{(\hat{x},x)} = S'[v']$;
    else
      $f(F_x, \ldots, F_{c'_x}) = (u', y') = S'[v']$; $f(F_{c_x}) = (u, y) = S[v]$;
      if $d(r, u') + w(f(F_x, \ldots, F_{c'_x})) + d(r, y') + d(r, q_x) - 2d(r, \texttt{lca}(y', q_x)) \leq$
      $d(r, u) + w(f(F_{c_x})) + d(r, y) + d(r, q'_x) - 2d(r, \texttt{lca}(y, q'_x))$
      then $f_{(\hat{x},x)} = f(F_x, \ldots, F_{c'_x})$; **else** $f_{(\hat{x},x)} = f(F_{c_x})$;
return $\{f_e \mid e \in E(T)\}$.

sets E_v and the arrangement of the tree vertices so as c_v is a vertex of the subtree of T rooted at the leftmost child of v, for every v, can be easily done in linear time. Finally, using Eq. (1), we have that the objective function value can be computed in constant time for every tree edge e and for every swap edge of e since essentially it only requires the computation of the least common ancestors (see [10]) and of the distances from the root to every vertex.

Therefore, to complete the proof, it remains to show that the number of operations performed on the two split-findmins $\mathcal{S}_1$ and $\mathcal{S}_2$, each storing n elements, is $O(m)$, and thus they will ask for $O(m \log \alpha(m, n))$ time (see [16]).

First, we bound the number of operations performed by the algorithm on $\mathcal{S}_1$. The number of decreasekey operations (see Step 1) is clearly upper bounded by $2m$ (one decreaseykey per each non-tree edge of G'). Let $Z_r = V(P(r, c_r)) \setminus \{r\}$ and, for every other vertex $v \neq r$, let $Z_v = V(P(v, c_v)) \setminus V(P(\hat{v}, c_{\hat{v}}))$. The number of split operations is equal to $\sum_{v \in V} |Z_v|$. As Lemma 2 implies that the set $\{Z_v \mid v \in V(T)\}$ is a partition of $V(T) \setminus \{r\}$, the number of split operations is $\sum_{v \in V(T)} |Z_v| = n - 1$. The number of findmin operations performed while visiting $x \neq r$ is equal to the size of L_x (see Step 3) plus 1 (see Step 4). As $|L_x|$ is upper bounded by the number of decreasekey operations performed by the algorithm on $\mathcal{S}_1$ while visiting e (see Step 1), plus $|Z_x|$ (see Step 2), the overall number of findmin operations is upper bounded by $\sum_{x \in V(T)} (|L_x| + 1) = 2m + n - 1 + n - 1 = 2(m + n - 1)$. Therefore, the overall number of split, findmin, and decreasekey operations performed by the algorithm on $\mathcal{S}_1$ is $2m + n - 1 + 2(m + n - 1) = 4m + 3(n - 1) = O(m)$.

Now, we bound the number of operations performed by the algorithm on $\mathcal{S}_2$. The number of decreasekey operations performed by the algorithm on $\mathcal{S}_2$ while visiting $x \neq r$ is upper bounded by $|L_x|$ (see Step 3). Therefore, the overall number of decreasekey operations performed by the algorithm on $\mathcal{S}_2$ is at most $2m + n - 1$. As the algorithm performs exactly one findmin and one split operation on $\mathcal{S}_2$ for every $e \in E(T)$ (see Step 4), the overall number of split, findmin, and decreasekey operations performed by the algorithm on $\mathcal{S}_2$ is $2m + 3(n - 1) = O(m)$. □

We end this section by showing that a modified version of Algorithm 1 can be used to solve the ABSE problem for the MDST.

Corollary 5 *The* ABSE *problem for the MDST can be solved in* $O(m \log \alpha(m, n))$ *time and* $O(m)$ *space.*

Proof Let P be a diametral path of the input tree T. In [8] it was proven that the problem of finding a best swap edge for each tree edge in $E(P)$ can be reduced in linear time to the problem of performing a sensitivity analysis of an MST. Furthermore, for the tree edges in $E(T) \setminus E(P)$, in [8] it is used a different implementation of the same algorithm we used to solve the `max`-ABSE problem by slightly modifying the primary key associated with $f = (u, y)$ to $\kappa_1(f) := w(P(u)) + w(f) + d(r, y)$, where $P(u)$ denotes a longest path in T starting at u. Since $P(v)$ can be computed in constant time for every v after a linear time and space preprocessing (see [8]), the claim follows from Theorem 1. □

4 A Faster Algorithm for sum-ABSE on Unweighted Graphs

In this section we solve efficiently the sum-ABSE problem on *unweighted* graphs. Thus, instead of talking about an SPT, we better will refer to a rooted *breadth-first tree* (BFT) T of G.

We start by refining the "verticalization" of non-tree edges described in Sect. 2. Let $f' = (\texttt{lca}(u, y), y)$ be a vertical edge associated with a non-tree edge $f = (u, y)$. Since G is unweighted and T is a BFT of G, we have that either of the three following cases arise for edge f: (i) $d(r, u) = d(r, y)$, or (ii) $d(r, u) = d(r, y) - 1$, or finally (iii) $d(r, u) = d(r, y) + 1$. Depending on which of the three cases applies, we have that f' is either of *type* (i), (ii) or (iii), respectively. Clearly, a symmetric argument applies to the other vertical edge $f'' = (\texttt{lca}(u, y), u)$ associated with f, and notice that if f' is of type (i), then the same holds for f'', while if f' is of type (ii), then f'' is of type (iii), and vice versa. Then, for any given vertex $v \in V(T)$, we can select a *representative* for any of these three types of vertical non-tree edges incident to v as follows: For each type, select a non-tree edge which swaps with a *largest number* of edges along the tree path from v to r. Notice that a representative edge is clearly preferable w.r.t. any other swap edge it was selected out of, since it allows for the same quality when swapping, while being usable longer. Thus, at most three vertical non-tree edges will remain associated with v, and all the other non-tree edges will be discarded. Observe that once again this refined preprocessing phase can be performed for all non-tree edges in $O(m)$ time, since it only requires to further classify vertical non-tree edges depending on their type, which is clearly doable in linear time. Notice that after the preprocessing, we have reduced to $O(n)$ the number of non-tree edges. This immediately allows to solve the sum-ABSE problem in linear space and $O\left(m + n\,\alpha(n, n)\log^2 n\right)$ time, by just using the fastest algorithm known for the weighted case [6]. Furthermore, Theorem 1 implies the following

Corollary 6 *The* max-ABSE *problem for the class of unweighted graphs can be solved in* $O\left(m + n \log \alpha(n, n)\right)$ *time and* $O(m)$ *space.* □

We show in the following a faster $O(m + n \log n)$ time algorithm for solving the sum-ABSE problem on unweighted graphs. Our algorithm runs in three phases and, in each phase, it considers only representative edges of either of three types. Indeed, for efficiency reasons, representative edges of different type need to be treated separately. This means that at each phase every vertex will have at most a *single* vertical edge associated with it, and at the end of a phase every tree edge will remain associated with a best swap edge of the current type. Finally, a best swap edge for a given tree edge will be selected as the best among those returned by the three phases. Hence, in the following we assume that all non-tree edges are of the same type. We define the *level* of a vertex v as $\ell(v) := d(r, v)$, and the *level* of a (vertical) non-tree edge (u, y) as the level of its lowest end vertex y. Furthermore, we define the *height* of v as $h(v) := \max\{\ell(v') \mid v' \in V(T_v)\}$.

For efficiency reasons that will be clearer later, our algorithm associates two keys with each non-tree edge f, and each of these keys will be separately managed by a suitable priority queue. The first key, say $\kappa_1(f)$, is a (constant) value which depends only on f, and that can be used to compare two competing swap edges of the *same*

level. Concerning the second key, this is instead a (non-constant) value which can be used to compare two competing swap edges of *different* levels. More precisely, such a key is a linear function of the form $\kappa_2(f,t) := at + b$, where a and b are constant values depending only on f, while t is a variable (called *virtual time*) that properly encodes the position in T of the failing edge e for which f must be evaluated.

The two keys of $f = (u, y)$ are defined as follows: $\kappa_1(f) := \sum_{v \in V(T)} d(y, v)$, while $\kappa_2(f,t) := 2\ell(y)t + \kappa_1(f) - \ell(y)n$. We can prove the following.

Lemma 5 *Let $e = (\hat{x}, x)$ be an edge of T and let f and f' be two swap edges of e. If $\kappa_2(f, |V(T_x)|) \leq \kappa_2(f', |V(T_x)|)$, then $\sum_{v \in V(T_x)} d_{e/f}(r, v) \leq \sum_{v \in V(T_x)} d_{e/f'}(r, v)$.*

Proof Let $f'' = (u'', y'')$ be any swap edge of e.

$$\begin{aligned}
\sum_{v \in V(T_x)} d_{e/f''}(r, v) &= \sum_{v \in V(T_x)} \big(d(r, u'') + w(f'') + d(y'', v)\big) \\
&= \big(d(r, u'') + w(f'')\big)\,|V(T_x)| + \sum_{v \in V(T)} d(y'', v) \\
&\quad - \sum_{v \in V(T) \setminus V(T_x)} d(y'', v) \\
&= \big(d(r, u'') + w(f'')\big)\,|V(T_x)| + \sum_{v \in V(T)} d(y'', v) \\
&\quad - \sum_{v \in V(T) \setminus V(T_x)} \big(d(y'', \hat{x}) + d(\hat{x}, v)\big) \\
&= \big(d(r, u'') + w(f'')\big)\,|V(T_x)| + \sum_{v \in V(T)} d(y'', v) \\
&\quad - (n - |V(T_x)|)\, d(y'', \hat{x}) - \sum_{v \in V(T) \setminus V(T_x)} d(\hat{x}, v) \\
&= \sum_{v \in V(T)} d(y'', v) - d(y'', \hat{x})n \\
&\quad + \big(d(r, u'') + w(f'') + d(y'', \hat{x})\big)\,|V(T_x)| \\
&\quad - \sum_{v \in V(T) \setminus V(T_x)} d(\hat{x}, v).
\end{aligned}$$

Now observe that $d(y'', \hat{x}) = \ell(y'') - \ell(\hat{x})$ and, by definition of $w(f'')$, that $d(r, u'') + w(f'') = \ell(y'') + k$, where k is a constant value depending only on the type of f''. Let

$$k' = \ell(\hat{x})(n - |V(T_x)|) + k|V(T_x)| - \sum_{v \in V(T) \setminus V(T_x)} d(\hat{x}, v).$$

Observe that k' depends only on e and on the type of f''. Therefore,

$$\begin{aligned}
\sum_{v \in V(T_x)} d_{e/f''}(r, v) &= \sum_{v \in V} d(y'', v) - \ell(y'')n + 2\ell(y'')|V(T_x)| + k' \\
&= 2\ell(y'')|V(T_x)| + \kappa_1(f'') - \ell(y'')n + k' \\
&= \kappa_2(f'', |V(T_x)|) + k'.
\end{aligned}$$

Since f and f' are of the same type,

$$\sum_{v \in V(T_x)} d_{e/f}(r, v) - \sum_{v \in V(T_x)} d_{e/f'}(r, v) = \kappa_2(f, |V(T_x)|) - \kappa_2(f', |V(T_x)|).$$

The claim follows. □

Furthermore, Lemma 5 immediately implies the following.

Corollary 7 *Let $e = (\hat{x}, x)$ be an edge of T and let f and f' be two swap edges of e of the same level. If $\kappa_1(f) \leq \kappa_1(f')$, then $\sum_{v \in V(T_x)} d_{e/f}(r, v) \leq \sum_{v \in V(T_x)} d_{e/f'}(r, v)$.*

Proof Since $f = (u, y)$ and $f' = (u', y')$ are of the same level, we have that $\ell(y) = \ell(y')$. Therefore, $\kappa_2(f, t) - \kappa_2(f', t) = 2\ell(y)t + \kappa_1(f) - \ell(y)n - 2\ell(y')t - \kappa_1(f') + \ell(y')n = \kappa_1(f) - \kappa_1(f')$. The claim follows. □

The main idea of our algorithm is to maintain efficiently a best swap edge for each of the levels below the failing edge. This will be done through the use of two types of priority queues, one for each type of key, according to their nature: a *Fibonacci heap* [7] to manage constant keys of the first type, and a *kinetic heap* [5] to manage variable keys of the second type.

A Fibonacci heap maintains a set of elements, where a key is associated with each element, and supports each of the N following operations in $O(\log N)$ time [4][8]:

`f-create`(): creates an empty Fibonacci heap and returns a pointer to it;
`f-insert`($\mathcal{F}, s, k$): if element s is not contained in the Fibonacci heap $\mathcal{F}$, then it adds s to $\mathcal{F}$ with a key value equal to k;
`f-delete`($\mathcal{F}, s$): deletes element s (given a pointer to it) and the key value associated with it from the Fibonacci heap $\mathcal{F}$;
`f-merge`($\mathcal{F}, \mathcal{F}'$): merges two distinct Fibonacci heaps $\mathcal{F}$ and $\mathcal{F}'$ and returns a pointer to the resulting heap;
`f-findmin`($\mathcal{F}$): returns the element of the Fibonacci heap $\mathcal{F}$ of minimum key.

A kinetic heap maintains a set of elements, where a linear function of the *virtual time* t is associated with each element, and supports the following operations:

`k-create`(): creates an empty kinetic heap and returns a pointer to it;
`k-insert`($\mathcal{K}, s, \phi(t)$): if element s is not contained in the kinetic heap $\mathcal{K}$, then it adds s to $\mathcal{K}$ and associates the linear function $\phi(t)$ to s;
`k-delete`($\mathcal{K}, s$): deletes element s (given a pointer to it) and the linear function associated with it from the kinetic heap $\mathcal{K}$;
`k-findmin`($\mathcal{K}, t_0$): returns the element of the kinetic heap $\mathcal{K}$ that minimizes the linear function associated with it w.r.t. the virtual time t_0.

Observe that kinetic heaps allow more sophisticated operations than Fibonacci heaps. However, findmin operations on kinetic heaps must satisfy a *monotonicity condition*, i.e., successive findmin operations must be performed with respect to non-decreasing

[8] More precisely, each creation, insertion, merge, and findmin operation requires only constant time.

values of the virtual time. As proved in [5], in an empty kinetic heap in which N operations are performed, each insertion and deletion requires $O(\log N)$ amortized time while each findmin operation requires $O(1)$ amortized time.

Now, we describe how the algorithm builds and maintains these heaps. In the initialization phase, for each vertex v of T different from r, the algorithm creates a Fibonacci heap, say $\mathcal{F}_{\ell(v)}(v)$, and, if v is a leaf of T, then it creates a kinetic heap $\mathcal{K}(v)$. Furthermore, for every non-tree edge $f = (u, y)$, the algorithm inserts f in $\mathcal{F}_{\ell(y)}(y)$ with a key value equal to $\kappa_1(f)$, and, if y is a leaf of T, then it inserts f in $\mathcal{K}(y)$ with a key value equal to $\kappa_2(f, t)$.

After the initialization phase, the algorithm considers the tree edges in a bottom-up fashion, by visiting the vertices of T in any postorder. When the algorithm visits vertex x (i.e., it considers the removal of edge $e = (\hat{x}, x)$ from T), it updates the two data structures so as the following invariants are maintained:

($\mathcal{I}_1$): for each level $j \in [\ell(x), h(x)]$, we have a Fibonacci heap $\mathcal{F}_j(x)$ containing all swap edges of e of level j, each with the corresponding key value equal to its primary key;
($\mathcal{I}_2$): there is a kinetic heap $\mathcal{K}(x)$ containing at most a swap edge of e for each level in the interval $[\ell(x), h(x)]$, with the property that $f = (u, y)$ is in $\mathcal{K}(x)$ and its key value is equal to $\kappa_2(f, t)$ iff f is a best possible swap edge of e among all swap edges of e of level $\ell(y)$.

Notice that from Corollary 7, ($\mathcal{I}_1$) allows to compute a best possible swap edge for e of level j via the call of `f-findmin`$\big(\mathcal{F}_j(x)\big)$. Furthermore, from Lemma 5, ($\mathcal{I}_2$) allows to compute a best swap edge f_e for e via the call of `k-findmin`$(\mathcal{K}(x), |V(T_x)|)$. As we will see, kinetic heaps are built on top of Fibonacci heaps whose use, in turn, will be only instrumental to keep low the number of more expensive operations performed on kinetic heaps.

Our algorithm proceeds as follows. If x is a leaf of T, then the algorithm computes a best swap edge of e by calling `k-findmin`$(\mathcal{K}(x), 1)$; otherwise, let $q_1, \ldots, q_s$ be the s children of x in T such that $h(q_1) \geq \cdots \geq h(q_s)$. The algorithm performs the following five steps (see also Fig. 4):

1. It inherits the kinetic heap and all Fibonacci heaps of q_1, i.e., it sets $\mathcal{K}(x) = \mathcal{K}(q_1)$ and $\mathcal{F}_j(x) = \mathcal{F}_j(q_1)$ for every $j \in [\ell(q_1), h(q_1)]$.[9]
2. If there is a swap edge f of e incident to x, then it inserts f in $\mathcal{K}(x)$ with a key value equal to $\kappa_2(f, t)$.
3. For each non-tree edge $f' = (u', y')$ with $u' = x$, it first deletes f' from the corresponding Fibonacci heap, say $\mathcal{F}_{\ell(y')}(q_k)$. Next it checks whether $\mathcal{K}(x)$ contains f'. If this is the case, then the algorithm: (i) deletes f' from $\mathcal{K}(x)$, (ii) performs an additional findmin on $\mathcal{F}_{\ell(y)}(q_k)$ to get a new best swap edge $f = (u, y)$ of level $\ell(y') = \ell(y)$, if any, and if this is the case (iii) inserts f into $\mathcal{K}(x)$ with a key value equal to $\kappa_2(f, t)$.
4. If $s \geq 2$, then for every level $j \in [\ell(x) + 1, h(q_2)]$, the algorithms does the following: It performs a findmin operation on $\mathcal{F}_j(q_1)$ to compute a best swap edge

[9] Notice that all these Fibonacci heaps can be inherited in $O(1)$ time by simply changing their reference from q_1 to x.

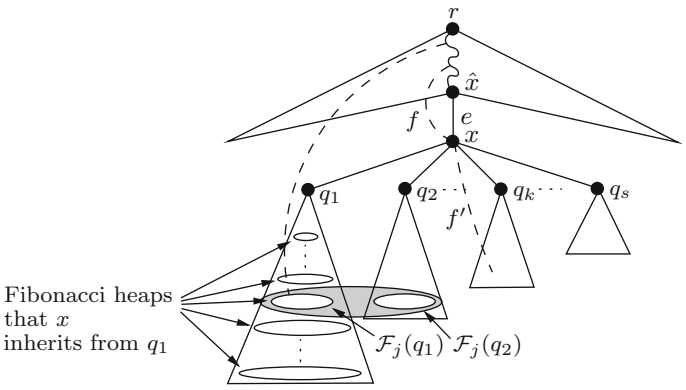

Fig. 4 An illustration of Steps 1–4 of the algorithm. The Kinetic heap $\mathcal{K}(x)$ is initialized to $\mathcal{K}(q_1)$. Edge f is inserted into $\mathcal{K}(x)$ while edge f' is deleted from the Fibonacci heap $F_\lambda(q_k)$, where λ is the level of f'

$\tilde{f}$ of (x, q_1) of level j, if any. Next, it updates $\mathcal{F}_j(x)$ by merging all Fibonacci heaps $\mathcal{F}_j(q_k)$, with $k = 2, \ldots, s$, to it; then, it performs a findmin operation on $\mathcal{F}_j(x)$ to compute a best possible swap edge $f = (u, y)$ of e of level j, if any. If $f \neq \tilde{f}$, then the algorithm removes $\tilde{f}$ from $\mathcal{K}(x)$ and afterwards it inserts f into $\mathcal{K}(x)$, with a key value equal to $\kappa_2(f, t)$.

5. Finally, the algorithm performs a findmin operation on $\mathcal{K}(x)$ with current virtual time $|V(T_x)|$ to compute a best swap edge of $e = (\hat{x}, x)$.

The pseudocode of the algorithm is given in Algorithm 2. This one uses a list $\mathcal{F}(v)$ of pointers to all Fibonacci heaps associated with v (a Fibonacci heap for each $j \in [\ell(v), h(v)]$), so as Fibonacci heaps can be inherited in $O(1)$ time. Furthermore, the algorithm uses two vectors F and K such that $F[f]$ and $K[f]$ are the pointers to the Fibonacci heap and the kinetic heap f belongs to, respectively.

Theorem 2 *Algorithm 2 solves the* sum-ABSE *problem in* $O(m + n \log n)$ *time and* $O(m)$ *space.*

Proof The correctness of the algorithm can be easily shown using induction w.r.t. the order in which the tree vertices are visited and by proving that invariants $(\mathcal{I}_1)$ and $(\mathcal{I}_2)$ are maintained at the end of the visit of x.

Concerning the time and space complexity of the algorithm, first of all, recall that the preprocessing step of the algorithm takes $O(m)$ time and space and generates an input instance with $O(n)$ non-tree edges. Moreover, in [6] it is shown that, after a linear time precomputation, we can compare any two competing swap edges and compute a value $\sum_{v' \in V(T)} d(v, v')$ for any vertex v of T in constant time. As a consequence, we can compute the primary and secondary key of any swap edge in constant time. Therefore, to derive the time complexity of the algorithm, we have to show that the number of operations performed on our data structures in each phase is $O(n)$. Indeed, Fibonacci heaps and kinetic heaps use $O(n)$ space as each non-tree edge appears in exactly one Fibonacci heap and one kinetic heap during the execution of the algorithm. Furthermore, in a Fibonacci heap, each creation, insertion, findmin, and merge takes

Algorithm 2: Algorithm for sum-ABSE.

Input : a two-edge-connected unweighted graph $G = (V(G), E(G))$, a root vertex $r \in V(G)$, and an SPT T of G rooted at r.

Output: a best swap edge f_e for every $e \in E(T)$ w.r.t. sum-ABSE.

Preprocess G so as explained at the beginning of the section;

let E^i, with $i = 1, 2, 3$, be the set of vertical non-tree edges of type i;
arrange T so as, for every non-leaf vertex v of T, the left-to-right order $q_1, \ldots, q_s$ of the s children of v is such that $h(q_1) \geq \cdots \geq h(q_s)$;
for *every* $i = 1, 2, 3$ **do**
  for *every vertex* $v \in V(T) \setminus \{r\}$ **do**
    $\mathcal{F}(v) =$ NULL; $\mathcal{F}_{\ell(v)}(v) =$ f-create(); add $\mathcal{F}_{\ell(v)}(v)$ to $\mathcal{F}(v)$;
    if v is a leaf of T **then** $\mathcal{K}(v) =$ k-create(); **else** $\mathcal{K}(v) =$ NULL;
  for *every* $f = (u, y) \in E^i$ **do**
    f-insert $\big(\mathcal{F}_{\ell(y)}(y), f, \kappa_1(f)\big)$; $F[f] = \mathcal{F}_{\ell(y)}(y)$; $K[f] =$ NULL;
    if *y is a leaf of* T **then**
      k-insert $(\mathcal{K}(y), f, \kappa_2(f, t))$; $K[f] = \mathcal{K}(y)$;
  for *every vertex x w.r.t. any fixed postorder visit of the tree vertices* **do**
    if *x is an inner vertex of* T **then**
      Let $q_1, \ldots, q_s$ be the children of x in T in a left-to-right order;
      append $\mathcal{F}(q_1)$ to $\mathcal{F}(x)$; $\mathcal{K}(x) = \mathcal{K}(q_1)$;
      if $f = (u, y) \in E^i$ *and* $y = x$ **then**
        k-insert $(\mathcal{K}(x), f, \kappa_2(f, t))$; $K[f] = \mathcal{K}(x)$;
      for *every* $f' \in \{(u, y) \in E^i \mid u = x\}$ **do**
        $\mathcal{F}' = F[f']$;
        f-delete $\big(\mathcal{F}', f'\big)$; $F[f'] =$ NULL;
        if $K[f'] \neq$ NULL **then**
          k-delete $\big(\mathcal{K}(f'), f'\big)$; $K[f'] =$ NULL;
          if $\mathcal{F}'$ *is not empty* **then**
            $f =$ f-findmin $\big(\mathcal{F}'\big)$;
            k-insert $(\mathcal{K}(x), f, \kappa_2(f, t))$; $K[f] = \mathcal{K}(x)$;
      if $s \geq 2$ **then**
        for *every* $j \in [\ell(x) + 1, h(q_2)]$ **do**
          if $\mathcal{F}_j(x)$ *is not empty* **then**
            $\tilde{f} =$ f-findmin$(\mathcal{F}_j(x))$;
            k-delete$\Big(\mathcal{K}(x), \tilde{f}\Big)$; $K(\tilde{f}) =$ NULL;
          for *every* $k \in \{2, \ldots, s\}$ **do**
            if $j \leq h(q_k)$ **then** $\mathcal{F}_j(x) =$ f-merge $\big(\mathcal{F}_j(x), \mathcal{F}_j(q_k)\big)$;
          if $\mathcal{F}_j(x)$ *is not empty* **then**
            $f =$ f-findmin$(\mathcal{F}_j(x))$;
            k-insert$(\mathcal{K}(x), f, \kappa_2(f, t))$; $K[f] = \mathcal{K}(x)$;
    if $\mathcal{K}(x)$ *is not empty* **then** $f_e^i =$ k-findmin $(K(x), |V(T_x)|)$; **else** $f_e^i = \perp$;

for every $e = (\hat{x}, x) \in E(T)$ **do** compute $f_e \in \arg \min_{f \in \{f_e^1, f_e^2, f_e^3\}} \sum_{v \in V(T_x)} d_{e/f}(r, v)$;

return $\{f_e \mid e \in E(T)\}$.

constant time, while each deletion takes $O(\log n)$ time [4]. Moreover, in a kinetic heap the amortized time for insert and delete operations is $O(\log n)$, while it is $O(1)$ for findmin operations [5].

In what follows, we bound the number of operations performed on our data structures for one execution of the outermost cycle of our algorithm. The total number of insert and merge operations on Fibonacci heaps is $O(n)$ since it is bounded by the initial number of Fibonacci heaps, namely $n-1$ (no Fibonacci heap is created and no element is inserted in Fibonacci heaps after the initialization phase). Therefore, the number of delete operations on Fibonacci heaps is also $O(n)$. The number of insertions and deletions on kinetic heaps is $O(n)$, since it is upper bounded by the number of leaves of T plus the number of merge and delete operations on Fibonacci heaps. Concerning the findmin operations on Fibonacci heaps, it is easy to see that we have $O(1)$ such operations for each merge of Fibonacci heaps and each deletion on kinetic heaps, which implies that they are $O(n)$. Clearly, we perform at most a single findmin on a kinetic heap for each edge of T for a total of $n-1$ findmin operations. Therefore, the overall number of operations performed on all Fibonacci heaps and kinetic heaps is $O(n)$. As the outermost for cycle is executed exactly three times the claim follows. □

5 Conclusions

In this paper, we have addressed the `max`-ABSE problem for the SPT of a non-negatively real-weighted graphs, and the `sum`-ABSE problem for the BFT of an unweighted graphs.

For the former problem, we provided an $O(m \log \alpha(m, n))$ time and linear space algorithm which can be easily adapted to solve the ABSE problem on an MDST with the same time and space complexity. In both cases, we improved onto the previous $O(m \log n)$ time algorithm [8]. The time complexity of our algorithm matches the time complexity of the best-known algorithm for the MST sensitivity analysis problem (see [16]) which, as observed in the introduction, can be reduced in linear time to our problem. As establishing the time complexity of the MST sensitivity analysis problem is regarded as one of the main open problems in the field of the design of efficient algorithms, any improvement in the time complexity of the `max`-ABSE problem would be a breakthrough.

For the latter problem, we first provided a sparsification technique that reduces the number of non-tree edges from $O(m)$ to $O(n)$, and then we designed an efficient $O(m + n \log n)$ time and linear space solving algorithm. Our algorithm uses the tree distances (which are integral numbers from 1 to n) to group non-tree edges w.r.t. the distance between their lower endvertices and the root of the tree so as a best swap edge of the failing tree edge in each group can be computed efficiently. For this reason, the extension of our algorithm to weighted graphs seems to be hard. Finally, it is worth noticing that the sparsification technique can also be used to reduce the size of the input instance of the `max`-ABSE problem on unweighted graphs to $O(n)$. As a consequence, the `max`-ABSE problem for the BFT of on unweighted graphs can be solved in $O(m + n \log \alpha(n, n))$ time.

As far as our future work on the topic is concerned, we believe that the study of ABSE problems for special classes of graphs (e.g., planar, Euclidean, etc.) is worthy of further investigation. Moreover, still in the spirit of analyzing the transient failure of each and every edge of a given tree-based structure, we plan to explore the relationship between the allowable number of structural changes to the original solution (which for a BSE problem is fixed to 1), and the quality of the obtained (approximate) recovery solution – as compared to a corresponding optimal replacement tree.

Acknowledgments This work was partially supported by the Research Grant PRIN 2010 "ARS Techno-Media" (Algorithms for Techno-Mediated Social Networks), funded by the Italian Ministry of Education, University, and Research.

References

1. Ackermann, W.: Zum hilbertschen aufbau der reellen zahlen. Math. Ann. **99**, 118–133 (1928)
2. Bilò, D., Gualà, L., Proietti, G.: Finding best swap edges minimizing the routing cost of a spanning tree. Algorithmica **68**(2), 337–357 (2014)
3. Bilò, D., Gualà, L., Proietti, G.: A faster computation of all the best swap edges of a shortest paths tree, Proceedings of the 21st Annual European Symposium on Algorithms (ESA'13), Sophia Antipolis, France, vol. 8125. Lecture Notes in Computer Science, pp. 157–168. Springer, Berlin (2013)
4. Brodal, G.S., Lagogiannis, G., Tarjan, R.E.: Strict Fibonacci heaps. Proceedings of the 44th Symposium on Theory of Computing (STOC'12), pp. 1177–1184. ACM Press (2012)
5. Brodal, G.S., Jacob, R.: Dynamic planar convex hull. Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS'02), pp. 617–626 . IEEE Computer Society (2002)
6. Di Salvo, A., 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)
7. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987)
8. Gfeller, B.: Faster swap edge computation in minimum diameter spanning trees. Algorithmica **62**(1–2), 169–191 (2012)
9. Gualà, L., Proietti, G.: Exact and approximate truthful mechanisms for the shortest-paths tree problem. Algorithmica **49**(3), 171–191 (2007)
10. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984)
11. Italiano, G.F., Ramaswami, R.: Maintaining spanning trees of small diameter. Algorithmica **22**(3), 275–304 (1998)
12. Ito, H., Iwama, K., Okabe, Y., Yoshihiro, T.: Polynomial-time computable backup tables for shortest-path routing. Proceedings of the 10th International Colloquium on Structural Information and Communication Complexity (SIROCCO'03), vol. 17. Proceedings in Informatics, Carleton Scientific, pp. 163–177 (2003)
13. Nardelli, E., Proietti, G., Widmayer, P.: How to swap a failing edge of a single source shortest paths tree. Proceedings of the 5th International Computing and Combinatorics Conference (COCOON'99), vol. 1627. Lecture Notes in Computer Science, pp. 144–153 . Springer, Berlin, (1999)
14. Nardelli, E., Proietti, G., Widmayer, P.: Finding all the best swaps of a minimum diameter spanning tree under transient edge failures. J. Graph Algorithms Appl. **5**(5), 39–57 (2001)
15. Nardelli, E., Proietti, G., Widmayer, P.: Swapping a failing edge of a single source shortest paths tree is good and fast. Algorithmica **36**(4), 361–374 (2003)
16. Pettie, S.: Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time. Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC'05), vol. 3827. Lecture Notes in Computer Science, pp. 964–973. Springer, Berlin (2005)
17. Proietti, G.: Dynamic maintenance versus swapping: an experimental study on shortest paths trees, Proceedings of the 3rd Workshop on Algorithm Engineering (WAE 2000), vol. 1982. Lecture Notes in Computer Science, pp. 207–217. Springer, Berlin (2000)

18. Tarjan, R.E.: Sensitivity analysis of minimum spanning trees and shortest path trees. Inf. Process. Lett. **14**(1), 30–33 (1982)
19. Wu, B.Y., Hsiao, C.-Y., Chao, K.-M.: The swap edges of a multiple-sources routing tree. Algorithmica **50**(3), 299–311 (2008)