# Sublinear-Time Maintenance of Breadth-First Spanning Tree in Partially Dynamic Networks

Monika Henzinger[1⋆], Sebastian Krinninger[1⋆], and Danupon Nanongkai[2⋆⋆]

[1] University of Vienna, Fakultät für Informatik, Austria
[2] Nanyang Technological University, Singapore

**Abstract.** We study the problem of maintaining a *breadth-first spanning tree* (BFS tree) in *partially dynamic* distributed networks modeling a sequence of either failures or additions of communication links (but not both). We show $(1+\epsilon)$-approximation algorithms whose amortized time (over some number of link changes) is *sublinear* in $D$, the *maximum diameter* of the network. This breaks the $\Theta(D)$ time bound of recomputing "from scratch".
Our technique also leads to a $(1+\epsilon)$-approximate incremental algorithm for single-source shortest paths (SSSP) in the sequential (usual RAM) model. Prior to our work, the state of the art was the classic *exact* algorithm of [17] that is optimal under some assumptions [45]. Our result is the first to show that, in the incremental setting, this bound can be beaten in certain cases if a small approximation is allowed.

## 1 Introduction

Complex networks are among the most ubiquitous models of interconnections between a multiplicity of individual entities, such as computers in a data center, human beings in society, and neurons in the human brain. The connections between these entities are constantly changing; new computers are gradually added to data centers, or humans regularly make new friends. These changes are usually *local* as they are known only to the entities involved. Despite their locality, they could affect the network *globally*; a single link failure could result in several routing path losses or destroy the network connectivity. To maintain its robustness, the network has to quickly respond to changes and repair its infrastructure. The study of such tasks has been the subject of several active areas of research, including dynamic, self-healing, and self-stabilizing networks.

One important infrastructure in distributed networks is the *breadth-first spanning (BFS) tree* [38, 42]. It can be used, for instance, to approximate the network diameter and to provide a communication backbone for broadcast of information through the network, routing, and control. In this paper, we study the problem of maintaining a BFS tree on dynamic distributed networks. Our main interest is repairing a BFS tree as fast as possible after each topology change.

⋆ The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 317532.
⋆⋆ Work partially done while at University of Vienna, Austria.

**Model.** We model the communication network by the CONGEST model [42], one of the major models of (locality-sensitive) distributed computation. Consider a synchronous network of processors modeled by an undirected unweighted graph $G$, where nodes model the processors and edges model the bounded-bandwidth links between the processors. We let $V(G)$ and $E(G)$ denote the set of nodes and edges of $G$, respectively. For any node $u$ and $v$, we let $d_G(u, v)$ be the distance between $u$ and $v$ in $G$. The processors (henceforth, nodes) are assumed to have unique IDs of $O(\log n)$ bits and infinite computational power. Each node has limited topological knowledge; in particular, it only knows the IDs of its neighbors and knows *no* other topological information (e.g., whether its neighbors are linked by an edge or not). The communication is synchronous and occurs in discrete pulses, called *rounds*. All the nodes wake up simultaneously at the beginning of each round. In each round each node $u$ is allowed to send an arbitrary message of $O(\log n)$ bits through each edge $(u, v)$ that is adjacent to $u$, and the message will reach $v$ at the end of the current round. There are several measures to analyze the performance of such algorithms, a fundamental one being the *running time*, defined as the worst-case number of rounds of distributed communication.

We model dynamic networks by a sequence of *attack* and *recovery* stages following the initial *preprocessing*. The dynamic network starts with a preprocessing on the initial network denoted by $N_0$, where nodes communicate on $N_0$ for some number of rounds. Once the preprocessing is finished, we begin the first attack stage where we assume that an adversary, who sees the current network $N_0$ and the states of all nodes, inserts and deletes an arbitrary number of edges in $N_0$. We denote the resulting network by $N_1$. This is followed by the first recovery stage where we allow nodes to communicate on $N_1$. After the nodes have finished communicating, the second attack stage starts, followed by the second recovery stage, and so on. We assume that $N_t$ is connected for every stage $t$. For any algorithm, we let the *total update time* be the total number of rounds needed by nodes to communicate during all recovery stages. Let the *amortized update time* be the total time divided by $q$ which is defined to be the number of edges inserted and deleted. Important parameters in analyzing the running time are $n$, the number of nodes (which remains the same throughout all changes) and $D$, the *maximum diameter*, defined to be the maximum diameter among all networks in $\{N_0, N_1, \ldots\}$. Note that $D \leq n$ since we assume that the network remains connected throughout. Following the convention from the area of (sequential) dynamic graph algorithms, we say that a dynamic network is *fully dynamic* if both insertions and deletions can occur in the attack stages. Otherwise, it is *partially dynamic*. Specifically, if only edge insertions can occur, it is an *incremental dynamic network*. If only edge deletions can occur, it is *decremental*.

Our model highlights two aspects of dynamic networks: (1) how quick a network can recover its infrastructure after changes (2) how edge failures and additions affect the network. These aspects have been studied earlier but we are not aware of any previous model identical to ours. To highlight these aspects, there are also a few assumptions inherent in our model. First, it is assumed that

the network remains static in each recovery stage. This assumption is often used (e.g. [29, 22, 32, 39]) and helps emphasizing the running time aspect of dynamic networks. Also note that we assume that the network is synchronous, but our algorithms will also work in an asynchronous model under the same asymptotic time bounds, using a synchronizer [42, 6]. Second, our model assumes that only edges can change. While there are studies that assume that nodes can change as well (e.g. [29, 22, 5]), this assumption is common and practical; see, e.g., [1, 7, 33, 15] and references therein. Our amortized update time is also similar in spirit to the amortized communication complexity heavily studied earlier (e.g. [7]). Finally, the results in this paper are on partially dynamic networks. While fully dynamic algorithms are more desirable, we believe that the partially dynamic setting is worth studying, for two reasons. The first reason, which is our main motivation, comes from an experience in the study of sequential dynamic algorithms, where insights from the partially dynamic setting often lead to improved fully dynamic algorithms. Moreover, partially dynamic algorithms can be useful in cases where one type of changes occurs much more frequently than the other type. For example, links constantly fail in physical networks, and it might not be necessary that the network has to be fixed (by adding a link) immediately. Instead, the network can try to maintain its infrastructures under a sequence of failures until the quality of service cannot be guaranteed, e.g. the network diameter becomes too large. Partially dynamic algorithms for maintaining a BFS tree, which in turns maintain the network diameter, are quite suitable for this type of applications.

**Problem.** We are interested in maintaining an approximate BFS tree. Our definition of approximate BFS trees below is a modification of the definition of BFS trees in [42, Definition 3.2.2].

**Definition 1 (Approximate BFS tree).** *For any $\alpha \geq 1$, an* $\alpha$-approximate BFS tree *of graph $G$ with respect to a given root $s$ is a spanning tree $T$ such that for every node $v$, $d_T(s, v) \leq \alpha d_G(s, v)$ (note that, clearly, $d_T(s, v) \geq d_G(s, v)$). If $\alpha = 1$, then $T$ is an* (exact) BFS tree.

The goal of our problem is to maintain an approximate BFS tree $T_t$ at the end of each recovery stage $t$ in the sense that every node $v$ knows its approximate distance to the preconfigured root $s$ in $N_t$ and, for each neighbor $u$ of $v$, $v$ knows if $u$ is its parent or child in $T_t$.

**Naive Algorithm.** As a toy example, observe that we can maintain a BFS tree simply by recomputing a BFS tree from scratch in each recovery stage. By using the standard algorithm (see, e.g., [42, 38]), we can do this in $\Theta(D_t)$, where $D_t$ is the diameter of graph $N_t$. Thus, the amortized time is $\Theta(D)$.

**Results.** Clearly, maintaining a BFS tree by recomputing it from scratch in every recovery stage requires $\Theta(D)$ time. Our main results are partially dynamic algorithms that break this time bound over a long run. They can maintain, for any constant $0 < \epsilon \leq 1$, a $(1+\epsilon)$-approximate BFS tree in time that is *sublinear in $D$* when amortized over $\omega(\frac{n \log D}{D})$ edge changes. To be precise, the amortized

update time over $q$ edge changes is

$$O\left(\left(1+\frac{n^{1/3}D^{2/3}}{\epsilon^{2/3}q^{1/3}}\right)\log D+n/q\right) \quad \text{and} \quad O\left(\left(\frac{n^{1/7}D^{6/7}}{q^{1/7}}\right)\log D+\frac{n}{\epsilon^{7/2}q}\right)$$

in the incremental and decremental setting, respectively. For the particular case of $q=\Omega(n)$, we get amortized update times of $O(D^{2/3}\log D)$ and $O(D^{6/7}\log D)$ for the incremental and decremental cases, respectively. Our algorithms do not require any prior knowledge about the dynamic network, e.g., $D$ and $q$. We note that, while there is no previous literature on this problem, one can parallelize the algorithm of Even and Shiloach [17] (also see [27, 45]) to obtain an amortized update time of $O(nD/q+1)$ over $q$ changes in both the incremental and the decremental setting. This bound is sublinear in $D$ when $q=\omega(n)$. Our algorithms give a sublinear time guarantee for a smaller number of changes, especially in applications where $D$ is large. Consider, for example, an application where we want to maintain a BFS tree of the network under link failures until the network diameter is larger than, say $n/10$ (at this point, the network will alert an administrator). In this case, our algorithms guarantee a sublinear amortized update time after a polylogarithmic number of failures while previous algorithms cannot do so.

In the sequential (usual RAM) model, our technique also gives an incremental $(1+\epsilon)$-approximation algorithm for the single-source shortest paths (SSSP) problem with an amortized update time of $O(mn^{2/5}/q^{3/5})$ per insertion and $O(1)$ query time, where $m$ is the number of edges in the final graph, and $q$ is the number of edge insertions. Prior to this result, only the classic exact algorithm of Even and Shiloach [17] from the 80s, with $O(mn/q)$ amortized update time, was known. No further progress has been made in the last three decades, and Roditty and Zwick [45] provided an explanation for this by showing that the algorithm of [17] is likely to be the fastest combinatorial exact algorithm, assuming that there is no faster combinatorial algorithm for Boolean matrix multiplication. Very recently, Bernstein and Roditty [8] showed that, in the decremental setting, this bound can be broken if a small approximation is allowed. Our result is the first one of the same spirit in the *incremental* setting; i.e., we brake the bound of Even and Shiloach for the case $q=o(n^{3/2})$.

**Related Work.** The problem of computing on dynamic networks is a classic problem in the area of distributed computing, studied from as early as the 70s; see, e.g. [7] and references therein. The main motivation is that dynamic networks better capture real networks, which experience failures and additions of new links. There is a large number of models of dynamic networks in the literature, each emphasizing different aspects of the problem. Our model closely follows the model of the sequential setting and, as discussed earlier, highlights the amortized update time aspect. It is closely related to the model in [31] where the main goal is to optimize the amortized update time using static algorithms in the recovery stages. The model in [31] is still slightly different from us in terms of allowed changes. For example, the model in [31] considers weighted networks and allows small weight changes but no topological changes; moreover, the message size can

be unbounded (i.e., the static algorithm in the recovery stage operates under the so-called LOCAL model). Another related model the *controlled dynamic model* (e.g. [30, 2]) where the topological changes do not happen instantaneously but are delayed until getting a permit to do so from the resource controller. Our algorithms can be used in this model as we can delay the changes until each recovery stage is finished. Our model is similar to, and can be thought of as a combination of, two types of models: those in, e.g., [29, 22, 32, 39] whose main interest is to determine how fast a network can recover from changes using static algorithms in the recovery stages, and those in, e.g., [7, 1, 15], which focus on the amortized cost per edge change. Variations of partially dynamic distributed networks have also been considered (e.g. [23, 44, 10, 9]).

The problem of constructing a BFS tree has been studied intensively in various distributed settings for decades (see [42, Chapter 5], [38, Chapter 4] and references therein). The studies were also extended to more sophisticated structures such as minimum spanning trees (e.g. [19, 34, 43, 14, 36, 37, 28, 13, 16]) and Steiner trees [26]. These studies usually focus on *static* networks, i.e., they assume that the network never changes and want to construct a BFS tree once, from scratch. While we are not aware of any results on maintaining a BFS tree on dynamic networks, there are a few related results. Much previous attention (e.g. [7]) has been paid on the problem of *maintaining a spanning tree*. In a seminal paper by Awerbuch et al. [7], it was shown that the amortized message complexity of maintaining a spanning tree can be significantly smaller than the cost of the previous approach of recomputing from scratch [1]. (A variant of their algorithm was later implemented as a part of the PARIS networking project at IBM [11] and slightly improved [35].) Our result is in the same spirit as [7] in breaking the cost of recomputing from scratch. An attempt to maintain spanning trees of small diameter has also motivated a problem called *best swap*. The goal is to replace a failed edge in the spanning tree by a new edge in such a way that the diameter is minimized. This problem has recently gained considerable attention in both sequential (e.g. [3, 24, 40, 41, 46, 25, 4, 12, 20]) and distributed (e.g. [21, 18]) settings.

In the sequential dynamic graph algorithms literature, a problem similar to ours is the single-source shortest paths (SSSP) problem on undirected graphs. This problem has been studied in partially dynamic settings and has applications to other problems, such as all-pairs shortest paths and reachability. As we have mentioned earlier, the classic bound of [17], which might be optimal [45], has recently been improved by a decremental approximation algorithm [8], and we achieve a similar result in the incremental setting.

## 2 Main Technical Idea

All our algorithms are based on a simple idea of *lazy updating*. Implementing this idea on different models requires modifications to cope with difficulties and to maximize efficiency. In this section, we explain the main idea by sketching a simple algorithm and its analysis for the incremental setting in the sequential

and the distributed model. We start with an algorithm that has *additive error*: Let $\kappa$ and $\delta$ be parameters. For every recovery stage $t$, we maintain a tree $T_t$ such that $d_{T_t}(s,v) \leq d_{N_t}(s,v) \leq d_{T_t}(s,v) + \kappa\delta$ for every node $v$. We will do this by recomputing a BFS tree from scratch for $O(q/\kappa + nD/\delta^2)$ times.

During the preprocessing, our algorithm constructs a BFS tree of $N_0$, denoted by $T_0$. This means that every node $u$ knows its parent and children in $T_0$ and the value of $d_{T_0}(s,u)$. Suppose that, in the first attack stage, an edge is inserted, say $(u,v)$ where $d_{N_0}(s,u) \leq d_{N_0}(s,v)$. As a result, the distances from $v$ to $s$ might decrease, i.e. $d_{N_1}(s,v) < d_{N_0}(s,v)$. In this case, the distances from $s$ to some other nodes (e.g. the children of $v$ in $T_0$) could decrease as well, and we may wish to recompute the BFS tree. Our approach is to do this *lazily*: We recompute the BFS tree only when the distance from $v$ to $s$ decreases by at least $\delta$; otherwise, we simply do nothing! In the latter case, we say that $v$ *is lazy*. Additionally, we regularly "clean up" by recomputing the BFS tree after every $\kappa$ insertions.

To prove an additive error of $\kappa\delta$, observe that errors occur for this single insertion only when $v$ is lazy. Intuitively, this causes an additive error of $\delta$ since we could have decreased the distance of $v$ and other nodes by at most $\delta$, but we did not. This argument can be extended to show that if we have $i$ lazy nodes, then the additive error will be at most $i\delta$. Since we do the cleanup every $\kappa$ insertions, the additive error will be at most $\kappa\delta$ as claimed.

For the number of BFS tree recomputations, first observe that the cleanup clearly contributes $O(q/\kappa)$ recomputations in total, over $q$ insertions. Moreover, a recomputation also could be caused by some node $v$, whose distance to $s$ decreases by at least $\delta$. Since every time a node $v$ causes a recomputation, its distance decreases by at least $\delta$, and $d_{N_0}(s,v) \leq D$, $v$ will cause the recomputation at most $D/\delta$ times. This naive argument shows that there are $nD/\delta$ recomputations (caused by $n$ different nodes) in total. This analysis is, however, *not* enough for our purpose. A tighter analysis, which is crucial to all our algorithms relies on the observation that when $v$ causes a recomputation, the distance from $v$'s neighbor, say $v'$, to $s$ also decreases by at least $\delta - 1$. Similarly, the distance of $v'$'s neighbor to $s$ decreases by at least $\delta - 2$, and so on. This leads to the conclusion that one recomputation corresponds to $(\delta + (\delta-1) + (\delta-2) + \ldots) = \Omega(\delta^2)$ distance decreases. Thus, the number of recomputations is at most $nD/\delta^2$. Combining the two bounds, we get that the number of BFS tree computations is $O(q/\kappa + nD/\delta^2)$ as claimed. We get the total time in sequential and distributed models by multiplying this number by $m$, the final number of edges, and $D$ (time for BFS tree computation), respectively.

To convert the additive error into a multiplicative error of $(1+\epsilon)$, we execute the above algorithm only for nodes whose distances to $s$ are greater than $\kappa\delta/\epsilon$. For other nodes, we can use the algorithm of Even and Shiloach [17] to maintain a BFS tree of depth $\kappa\delta/\epsilon$. This requires an additional time of $O(m\kappa\delta/\epsilon)$ in the sequential model and $O(n\kappa\delta/\epsilon)$ in the distributed model.

By setting $\kappa$ and $\delta$ appropriately, the above algorithm immediately gives us the claimed time bound for the sequential model. For incremental distributed networks, we need one more idea called *layering*, where we use different values of

$\delta$ and $\kappa$ depending on the distance of each node to $s$. In the decremental setting, the situation is much more difficult, mainly because it is nearly impossible for a node $v$ to determine how much its distance to $s$ has increased after a deletion. Moreover, unlike the incremental case, nodes cannot simply "do nothing" when an edge is deleted. We have to cope with this using several other ideas, e.g., constructing an imaginary tree (where edges sometimes represent paths).

## 3 Incremental Algorithm

We now present a framework for an incremental algorithm that allows up to $q$ edge insertions and provides an additive approximation of the distances to a distinguished node $s$. Subsequently we will explain how to use this algorithm to get $(1+\epsilon)$-approximations in the RAM model and the distributed model, respectively. For simplicity we assume that the initial graph is connected. The algorithm can be modified to work without this assumption within the same running time. We defer the details to a full version of the paper.

The algorithm (see Algorithm 1) works in *phases*. At the beginning of every phase we compute a BFS tree $T_0$ of the current graph, say $G_0$. Every time an edge $(u, v)$ is inserted, the distances of some nodes to $s$ in $G$ might decrease. Our algorithm tries to be *as lazy as possible*. That is, when the decrease does not exceed some parameter $\delta$, our algorithm keeps its tree $T_0$ and accepts an *additive error* of $\delta$ for every node. When the decrease exceeds $\delta$, our algorithm starts a new phase and recomputes the BFS tree. It also start a new phase after every $\kappa$ edge insertions to keep the additive error limited to $\kappa\delta$. The algorithm will answer a query for the distance from a node $x$ to $s$ by returning $d_{G_0}(x, s)$, the distance from $x$ to $s$ at the beginning of the current phase. It can also return the path from $x$ to $s$ in $T_0$ of length $d_{G_0}(x, s)$. Besides $\delta$ and $\kappa$, the algorithm has a third parameter $X$ which indicates up to which distance from $s$ the BFS tree will be computed. In the following we denote by $G_0$ the state of the graph at the beginning of the current phase and by $G$ we denote the current state of the graph after all insertions. It is easy to see that the algorithm gives the desired additive

**Algorithm 1** Incremental algorithm

```
1: procedure INSERT(u, v)
2:     k ← k + 1
3:     if k = κ then INITIALIZE( )
4:     if d_{G_0}(u, s) > d_{G_0}(v, s) + δ then INITIALIZE( )
5: procedure INITIALIZE( )      (Start new phase)
6:     k = 0
7:     Compute BFS tree T of depth X rooted at s and current distances d_{G_0}(·, s)
```

approximation by considering the shortest path of a node $x$ to the root $s$ in the current graph $G$. By the main rule in Line 4 of the algorithm, the inequality $d_{G_0}(u, s) \leq d_{G_0}(v, s) + \delta$ holds for every edge $(u, v)$ that was inserted since the

beginning of the current phase (otherwise a new phase would have been started). Since at most $\kappa$ edges have been inserted, the additive error is at most $\kappa\delta$.

**Lemma 2.** *For every node $x$ such that $d_{G_0}(x,s) \leq X$, Algorithm 1 maintains the invariant $d_G(x,s) \leq d_{G_0}(x,s) \leq d_G(x,s) + \kappa\delta$.*

If we insert an edge $(u,v)$ such that the inequality $d_{G_0}(u,s) \leq d_{G_0}(v,s) + \delta$ does not hold, we cannot guarantee the additive error anymore. Nevertheless the algorithm makes a lot of progress in some sense: After the edge insertion, $u$ is linked to $v$ whose initial distance to $s$ was significantly smaller than the one from $u$ to $s$. This implies that the distance from $u$ to $s$ has decreased by at least $\delta$ since the beginning of the current phase.

**Lemma 3.** *If an edge $(u,v)$ is inserted such that $d_{G_0}(u,s) > d_{G_0}(v,s)+\delta$, then $d_{G_0}(u,s) \geq d_G(u,s) + \delta$.*

Since we consider undirected, unweighted graphs, a large decrease in distance for one node also implies a large decrease in distance for many other nodes.

**Lemma 4.** *Let $G$ and $G'$ be unweighted, undirected graphs such that $G$ is connected, $V(G) = V(G')$, and $E(G) \subseteq E(G')$. If there is a node $y$ such that $d_G(y,s) \geq d_{G'}(y,s) + \delta$, then $\sum_{x \in V(G)} d_G(x,s) \geq \sum_{x \in V(G')} d_{G'}(x,s) + \Omega(\delta^2)$.*

This is the key observation for the efficiency of our algorithm as it limits the number of times a new phase starts, which is the expensive part of our algorithm.

**Lemma 5.** *If $\kappa \leq q$ and $\delta \leq X$, then the total running time of Algorithm 1 is $O(T_{BFS}(X) \cdot q/\kappa + T_{BFS}(X) \cdot nX/\delta^2 + q)$ where $T_{BFS}(X)$ is the time needed for computing a BFS tree up to depth $X$.*

### 3.1 RAM model

In the RAM model we use a standard approach for turning the additive $\kappa\delta$-approximation of Algorithm 1 into a multiplicative $(1+\epsilon)$-approximation. We use the Even-Shiloach algorithm for maintaining the *exact* distances to $s$ up to depth $\kappa\delta/\epsilon$. Algorithm 1 now only has to be approximately correct for every node $x$ such that $d_G(x,s) \geq \kappa\delta/\epsilon$, and indeed $d_G(x,s) + \kappa\delta \leq d_G(x,s) + \epsilon d_G(x,s) = (1+\epsilon)d_G(x,s)$. The Even-Shiloach tree adds $O(m\kappa\delta/\epsilon)$ to the running time of Lemma 5 (where computing a BFS tree takes time $O(m)$). By setting

$$\kappa = q^{3/5}/n^{2/5} \quad \text{and} \quad \delta = n^{4/5}/q^{1/5},$$

we get the following theorem.

**Theorem 6.** *In the RAM model, there is an incremental $(1+\epsilon)$-approximate SSSP algorithm for inserting up to $q$ edges that has a total update time of $O(mn^{2/5}q^{2/5}/\epsilon)$ where $m$ is the number of edges in the final graph. It answers distance and path queries in optimal worst-case time.*

### 3.2 Distributed model

In the distributed model, we use a different approach for obtaining the $(1+\epsilon)$-approximation. We run $\log D$ parallel instances of Algorithm 1, where instance $i$ provides a $(1+\epsilon)$-approximation for nodes in the distance range from $2^i$ to $2^{i+1}$. For every such range, we can determine the targeted additive approximation $A$ that guarantees a $(1+\epsilon)$-approximation in that range. We set the parameters $\kappa$ and $\delta$ in a way that gives $\kappa\delta = A$ and minimizes the running time of Algorithm 1. Here, this approach is more efficient than the two-layered approach that uses a single Even-Shiloach tree for small distances. The fact that the time for computing a BFS tree of depth $X$ is proportional to $X$ in the distributed model nicely fits into the running time of the multi-layer approach.

**Theorem 7.** *In the distributed model, there is an incremental algorithm for maintaining a $(1+\epsilon)$-approximate BFS tree under up to $q$ edge insertions that has a total running time of $O((q^{2/3}n^{1/3}D^{2/3}/\epsilon^{2/3}+q)\log D+n)$, where $D$ is the initial diameter.*

## 4 Decremental Algorithm

In the decremental setting we use an algorithm of the same flavor as in the incremental setting (see Algorithm 2). However, the update procedure is more complicated because it is not obvious which edge should be used to repair the tree after a deletion. Our solution exploits the fact that in the distributed model it is relatively cheap to examine the local neighborhood of every node. As in the incremental setting, the algorithm has the parameters $\kappa$, $\delta$, and $X$. The tree update procedure of Algorithm 2 either computes a (weighted) tree $T$ that approximates the real distances with additive error $\kappa\delta$, or it reports a distance increase by at least $\delta$ since the beginning of the current phase. Let $T_0$ denote the BFS tree computed at the beginning of the current phase and let $F$ be the forest resulting from removing those edges from $T_0$ that have already been deleted in the current phase. After every edge deletion, the tree update procedure tries to rebuild a tree $T$ by starting from $F$. Every node $u$ that had a parent in $T_0$ but has no parent in $F$ tries to find a "good" node $v$ to reconnect to. This process is repeated until $F$ is a full tree again. Algorithm 2 imposes three conditions (Lines 17-19) on a "good" node $v$. Condition (1) guarantees that the error introduced by each reconnection is at most $\delta$, (2) avoids that the reconnections introduce any cycles, and (3) states that the node $v$ should be found relatively close to $u$. This is the key to efficiently find such a node.

### 4.1 Analysis of Tree Update Procedure

For the analysis of the tree update procedure of Algorithm 2, we assume that the edges in $F$ are directed. When we compute a BFS tree, we consider all edges as directed towards the root. The *weighted*, *directed* distance from $x$ to $y$ in $F$ is denoted by $d^w_F(x,y)$. We assume for the analysis that $F$ initially contains *all*

**Algorithm 2** Decremental algorithm

**procedure** DELETE($u$, $v$)
  $k \leftarrow k + 1$
  **if** $k = \kappa$ **then** INITIALIZE( )
  Delete edge $(u, v)$ from $F$
  Compute tree $T$ from $F$ and $d_{G_0}(\cdot, s)$: $T \leftarrow$ UPDATETREE($F, d_{G_0}(\cdot, s)$)
  **if** UPDATETREE reports distance increase by at least $\delta$ **then** INITIALIZE( )
**procedure** INITIALIZE( )  *(Start new phase)*
  $k = 0$
  Compute BFS tree $T$ of depth $X$ rooted at $s$. Set $F \leftarrow T$
  Compute current distances $d_{G_0}(\cdot, s)$
**procedure** UPDATETREE($F$, $d_{G_0}(\cdot, s)$)
  *At any time:* $U = \{u \in V \mid u$ *has no outgoing edge in* $F$ *and* $u \neq s\}$
  **while** $U \neq \emptyset$ **do**
    **for all** $u \in U$ **do** $u$ marks every node $x$ such that $d^w_F(x, u) \leq 3\kappa\delta$
    **for all** $u \in U$ **do**  *(Search process)*
      $u$ tries to find a node $v$ by breadth-first search such that
      (1) $d_G(u, v) + d_{G_0}(v, s) \leq d_{G_0}(u, s) + \delta$
      (2) $v$ is not marked
      (3) $d_G(u, v) \leq (\kappa + 1)\delta + 1$
      **if** such a node $v$ could be found **then**
        Add edge $(u, v)$ of weight $d^w_F(u, v) = d_G(u, v)$ to $F$
    **if** no such node $v$ could be found for any $u \in U$ **then**
      **return** "distance increase by at least $\delta$"
  **return** $F$

nodes. By $T$ we denote the graph returned by the algorithm. By Condition (1), every reconnection made by the tree update procedure adds an additive error of $\delta$. In total there are at most $\kappa$ reconnections (one per previous edge deletion) and therefore the total additive error introduced is $\kappa\delta$.

**Lemma 8.** *After every iteration, we have, for all nodes* $x$ *and* $y$ *such that* $d^w_F(x, y) < \infty$, $d^w_F(x, y) + d_{G_0}(y, s) \leq d_{G_0}(x, s) + \kappa\delta$.

By Condition (2) we avoid that the reconnection process introduces any cycle. Ideally, a node $u \in U$ should reconnect to a node $v$ that is in the subtree of the root $s$. We could achieve this if every node in $U$ marked its whole subtree in $F$. However, this would be too inefficient. Instead, marking the subtree up to a limited depth ($3\kappa\delta$) is sufficient.

**Lemma 9.** *After every iteration, the graph* $F$ *is a forest.*

We can show that the algorithm makes progress in every iteration. There is always at least one node for which a "good" reconnection is possible that fulfills the conditions of the algorithm. Even more, if such a node does not exist, then there is a node whose distance to $s$ has increased by at least $\delta$ since the beginning of the current phase.

**Lemma 10.** *In every iteration, if $d_G(x,s) \leq d_{G_0}(x,s) + \delta$ for every node $x \in U$, then for every node $u \in U$ with minimal $d_{G_0}(u,s)$, there is a node $v \in V$ such that (1) $d_G(u,v) + d_{G_0}(v,s) \leq d_{G_0}(u,s) + \delta$, (2) $v$ is not marked, and (3) $d_G(u,v) \leq (\kappa+1)\delta + 1$.*

The marking and the search process both take time $O(\kappa\delta)$. Since there is at least one reconnection in every iteration (unless the algorithm reports a distance increase), there are at most $\kappa$ iterations that take time $O(\kappa\delta)$ each.

**Lemma 11.** *The tree update procedure of Algorithm 2 either reports "distance increase" and guarantees that there is a node $x$ such that $d_G(x,s) > d_{G_0}(x,s)+\delta$, or it computes a tree $T$ such that for every node $x$ we have $d_{G_0}(x,s) \leq d_G(x,s) \leq d_T^w(x,s) \leq d_{G_0}(x,s) + \kappa\delta$. It runs in time $O(\kappa^2\delta)$.*

In the following we clarify some implementation issues of the tree update procedure in the distributed model.

**Weighted Edges.** The tree computed by the algorithm contains weighted edges. Such an edge $e$ corresponds to a path $P$ of the same distance in the network. We implement weighted edges by a routing table for every node $v$ that stores the next node on $P$ if a message is sent over $v$ as part of the weighted edge $e$.

**Avoiding Congestion.** The marking can be done in parallel without congestion because the trees in the forest $F$ do not overlap. We avoid congestion during the search as follows. If a node receives more than one message from its neighbors, we always give priority to search requests originating from the node $u$ with the lowest distance $d_{G_0}(u,s)$ to $s$ in $G_0$. By Lemma 10, we know that the search of at least one of the nodes $u$ with minimal $d_{G_0}(u)$ will always be successful.

**Reporting Deletions.** The nodes that do not have a parent in $F$ before the update procedure starts do not necessarily know that a new edge deletion has happened. Such a node only has to become active and do the marking and the search if there is a change in its neighborhood of distance $3\kappa\delta$, otherwise it can still use the weighted edge in the tree $T$ that it previously used because the three conditions imposed by the algorithm will still be fulfilled. After the deletion of an edge $(x,y)$, the nodes $x$ and $y$ can inform all nodes at distance $3\kappa\delta$ in time $O(\kappa\delta)$, which is within our projected running time.

### 4.2 Analysis of Decremental Distributed Algorithm

The tree update procedure provides an additive approximation of the shortest paths. It also provides a means for detecting that the distance of some node to $s$ has increased by at least $\delta$ since the beginning of the current phase. Using the distance increase argument of Lemma 4, we can provide a running time analysis for the decremental algorithm that is very similar to the incremental algorithm.

**Lemma 12.** *If $q \leq \kappa$ and $\delta \leq X$, then the total running time of Algorithm 2 is $O(qX/\kappa + nX^2/\delta^2 + q\kappa^2\delta)$. It maintains the following invariant: If $d_{G_0}(x,s) \leq X$, then $d_{G_0}(x,s) \leq d_G(x,s) \leq d_T^w(x,s) \leq d_{G_0}(x,s) + \kappa\delta$.*

We use a similar approach as in the incremental setting to get the $(1+\epsilon)$-approximation. We run $i$ parallel instances of the algorithm where each instance covers the distance range from $2^i$ to $2^{i+1}$. By a careful choice of the parameters $\kappa$ and $\delta$ for each instance we can guarantee a $(1+\epsilon)$-approximation.

**Theorem 13.** *In the decremental distributed dynamic network, we can maintain a $(1+\epsilon)$-approximate BFS tree over $q$ edge deletions in a total running time of $O((q^{6/7}n^{1/7}D^{6/7})\log D + (n+q)/\epsilon^{7/2})$, where $D$ is the maximum diameter.*

## 5 Conclusion and Open Problems

In this paper, we showed that maintaining a breadth-first search spanning tree can be done in an amortized time that is sublinear in $D$ in partially dynamic networks. Many problems remain open. For example, can we get a similar result for the case of *fully dynamic* networks? How about *weighted* networks (even partially dynamic ones)? Can we also get a sublinear time bound for the *all-pairs shortest paths* problem? Moreover, in addition to the sublinear time complexity achieved in this paper, it is also interesting to obtain algorithms small bounds on message complexity and memory.

**Acknowledgements.** We thank the reviewers of ICALP 2013 for pointing to related papers and to an error in an example given in the previous version.

## References

1. Afek, Y., Awerbuch, B., Gafni, E.: Applying static network protocols to dynamic networks. In: FOCS. pp. 358–370 (1987)
2. Afek, Y., Awerbuch, B., Plotkin, S.A., Saks, M.E.: Local management of a global resource in a communication network. J. ACM 43(1), 1–19 (1996)
3. Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M.: Minimizing diameters of dynamic trees. In: ICALP. pp. 270–280 (1997)
4. 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), announced at ICALP'97 and SWAT'00
5. Augustine, J., Pandurangan, G., Robinson, P., Upfal, E.: Towards robust and efficient computation in dynamic peer-to-peer networks. In: SODA. pp. 551–569 (2012)
6. Awerbuch, B.: Complexity of network synchronization. J. ACM 32(4), 804–823 (1985)
7. Awerbuch, B., Cidon, I., Kutten, S.: Optimal maintenance of a spanning tree. J. ACM 55(4) (2008), announced at FOCS'90
8. Bernstein, A., Roditty, L.: Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In: SODA. pp. 1355–1365 (2011)
9. Cicerone, S., D'Angelo, G., Stefano, G.D., Frigioni, D.: Partially dynamic efficient algorithms for distributed shortest paths. Theor. Comput. Sci. 411(7-9), 1013–1037 (2010)
10. Cicerone, S., D'Angelo, G., Stefano, G.D., Frigioni, D., Petricola, A.: Partially dynamic algorithms for distributed shortest paths and their experimental evaluation. JCP 2(9), 16–26 (2007)

11. Cidon, I., Gopal, I.S., Kaplan, M.A., Kutten, S.: A distributed control architecture of high-speed networks. IEEE Transactions on Communications 43(5), 1950–1960 (1995)
12. Das, S., Gfeller, B., Widmayer, P.: Computing all best swaps for minimum-stretch tree spanners. J. Graph Algorithms Appl. 14(2), 287–306 (2010)
13. Das Sarma, A., Holzer, S., Kor, L., Korman, A., Nanongkai, D., Pandurangan, G., Peleg, D., Wattenhofer, R.: Distributed verification and hardness of distributed approximation. SIAM J. Comput. 41(5), 1235–1265 (2012), announced at STOC'11
14. Elkin, M.: A faster distributed protocol for constructing a minimum spanning tree. J. Comput. Syst. Sci. 72(8), 1282–1308 (2006), announced at SODA'04
15. Elkin, M.: A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In: PODC. pp. 185–194 (2007)
16. Elkin, M., Klauck, H., Nanongkai, D., Pandurangan, G.: Quantum distributed network computing: Lower bounds and techniques. CoRR abs/1207.5211 (2012)
17. Even, S., Shiloach, Y.: An on-line edge-deletion problem. J. ACM 28(1), 1–4 (1981)
18. Flocchini, P., Enriques, A.M., Pagli, L., Prencipe, G., Santoro, N.: Point-of-failure shortest-path rerouting: Computing the optimal swap edges distributively. IEICE Transactions 89-D(2), 700–708 (2006)
19. Garay, J., Kutten, S., Peleg, D.: A sublinear time distributed algorithm for minimum-weight spanning trees. SIAM J. on Computing 27, 302–316 (1998), announced at FOCS'93
20. Gfeller, B.: Faster swap edge computation in minimum diameter spanning trees. Algorithmica 62(1-2), 169–191 (2012), announced at ESA'08
21. Gfeller, B., Santoro, N., Widmayer, P.: A distributed algorithm for finding all best swap edges of a minimum-diameter spanning tree. IEEE Trans. Dependable Sec. Comput. 8(1), 1–12 (2011), announced at DISC'07
22. Hayes, T.P., Saia, J., Trehan, A.: The forgiving graph: a distributed data structure for low stretch under adversarial attack. Distributed Computing 25(4), 261–278 (2012), announced at PODC'09
23. Italiano, G.F.: Distributed algorithms for updating shortest paths. In: WDAG(DISC). pp. 200–211 (1991)
24. Italiano, G.F., Ramaswami, R.: Maintaining spanning trees of small diameter. Algorithmica 22(3), 275–304 (1998), announced at ICALP'94
25. Ito, H., Iwama, K., Okabe, Y., Yoshihiro, T.: Single backup table schemes for shortest-path routing. Theor. Comput. Sci. 333(3), 347–353 (2005)
26. Khan, M., Kuhn, F., Malkhi, D., Pandurangan, G., Talwar, K.: Efficient distributed approximation algorithms via probabilistic tree embeddings. Distributed Computing 25(3), 189–205 (2012), announced at PODC'08
27. King, V.: Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In: FOCS. pp. 81–91 (1999)
28. Kor, L., Korman, A., Peleg, D.: Tight bounds for distributed MST verification. In: STACS. pp. 69–80 (2011)
29. Korman, A.: Improved compact routing schemes for dynamic trees. In: PODC. pp. 185–194 (2008)
30. Korman, A., Kutten, S.: Controller and estimator for dynamic networks. Inf. Comput. 223, 43–66 (2013)
31. Korman, A., Peleg, D.: Dynamic routing schemes for graphs with low local density. ACM Transactions on Algorithms 4(4) (2008)
32. Krizanc, D., Luccio, F.L., Raman, R.: Compact routing schemes for dynamic ring networks. Theory Comput. Syst. 37(5), 585–607 (2004)

33. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: STOC. pp. 513–522 (2010)
34. Kutten, S., Peleg, D.: Fast distributed construction of small $k$-dominating sets and applications. J. Algorithms 28(1), 40–66 (1998), announced at PODC'95
35. Kutten, S., Porat, A.: Maintenance of a spanning tree in dynamic networks. In: DISC. pp. 342–355 (1999)
36. Lotker, Z., Patt-Shamir, B., Peleg, D.: Distributed MST for constant diameter graphs. Distributed Computing 18(6), 453–460 (2006), announced at PODC'01
37. Lotker, Z., Pavlov, E., Patt-Shamir, B., Peleg, D.: MST construction in $o(\log \log n)$ communication rounds. In: SPAA. pp. 94–100 (2003)
38. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, San Francisco, CA, USA (1996)
39. Malpani, N., Welch, J.L., Vaidya, N.H.: Leader election algorithms for mobile ad hoc networks. In: DIAL-M. pp. 96–103 (2000)
40. 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)
41. Nardelli, E., Proietti, G., Widmayer, P.: Swapping a failing edge of a single source shortest paths tree is good and fast. Algorithmica 35(1), 56–74 (2003)
42. Peleg, D.: Distributed computing: a locality-sensitive approach. SIAM, Philadelphia, PA, USA (2000)
43. Peleg, D., Rubinovich, V.: A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. SIAM J. Comput. 30(5), 1427–1442 (2000), announced at FOCS'99
44. Ramarao, K.V.S., Venkatesan, S.: On finding and updating shortest paths distributively. J. Algorithms 13(2), 235–257 (1992)
45. Roditty, L., Zwick, U.: On dynamic shortest paths problems. Algorithmica 61(2), 389–401 (2011), announced at ESA'04
46. Salvo, A.D., Proietti, G.: Swapping a failing edge of a shortest paths tree by minimizing the average stretch factor. In: SIROCCO. pp. 99–110 (2004)

# Appendix

## A Omitted Proofs of Section 3

**Theorem 14.** *The algorithm of Even and Shiloach is a partially dynamic algorithm for maintaining single-source shortest paths up to depth $X$ under edge insertions (deletions) in an unweighted, undirected graph with a total running time of $O(mX)$ in the RAM model and $O(nX + q)$ in the distributed model where $q$ is the number of insertions (deletions).*

**Lemma 2 (Restated).** *For every node $x$ such that $d_{G_0}(x, s) \leq X$ Algorithm 1 maintains the invariant*

$$d_G(x, s) \leq d_{G_0}(x, s) \leq d_G(x, s) + \kappa\delta\,.$$

*Proof.* The algorithm can only maintain the invariant for every node $x$ such that $d_{G_0}(x,s) \leq X$ because other nodes are not contained in the BFS tree of the current phase. It is clear that $d_G(x,s) \leq d_{G_0}$ because $G$ is the result of inserting edges into $G_0$. In the following we argue about the second inequality.

Consider the shortest path $P = x_l, x_{l-1}, \dots x_0$ of length $l$ from $x$ to $s$ in $G$ (where $x_l = x$ and $x_0 = s$). Let $S_j$ (with $0 \leq j \leq l$) denote the number of edges in the subpath $x_j, x_{j-1}, \dots, x_0$ that were inserted since the beginning of the current phase.

*Claim.* For every integer $j$ with $0 \leq j \leq l$ we have

$$d_{G_0}(x_j, s) \leq d_G(x_j, s) + S_j\delta$$

Clearly the claim already implies the inequality we want to prove since there are at most $\kappa$ edges that have been inserted since the beginning of the current phase which gives the following chain of inequalities:

$$d_{G_0}(x,s) = d_{G_0}(x_l, s) \leq d_G(x_l, s) + S_l\delta \leq d_G(x,s) + \kappa\delta$$

Now we proceed with the inductive proof of the claim The induction base $j = 0$ is trivially true because $x_j = s$. Now consider the induction step where we assume that the inequality holds for $j$ and we have to show that it also holds for $j+1$.

Consider first the case that the edge $(x_{j+1}, x_j)$ is one of the edges that have been inserted since the beginning of the current phase. By the rule of the algorithm we know that $d_{G_0}(x_{j+1}, s) \leq d_{G_0}(x_j, s)+\delta$ and by the induction hypothesis we have $d_{G_0}(x_j, s) \leq d_G(x_j, s)+S_j\delta$. By combining these two inequalities we get $d_{G_0}(x_{j+1}, s) \leq d_G(x_j, s) + (S_j+1)\delta$. The desired inequality now follows because $S_{j+1} = S_j + 1$ and because $d_G(x_j, s) \leq d_G(x_{j+1}, s)$ (on the shortest path $P$, $x_j$ is closer to $s$ than $x_{j+1}$).

Now consider the case that the edge $(x_{j+1}, x_j)$ is not one of the edges that have been inserted since the beginning of the current phase. In that case the edge $(x_{j+1}, x_j)$ in contained in the graph $G_0$ and thus $d_{G_0}(x_{j+1}, s) \leq d_{G_0}(x_j, s) + 1$. By the induction hypothesis we have $d_{G_0}(x_j, s) \leq d_G(x_j, s)+S_j\delta$. By combining these two inequalities we get $d_{G_0}(x_{j+1}, s) \leq d_G(x_j, s)+1+S_j\delta$. Since $x_{j+1}$ and $x_j$ are neighbours on the shortest path $P$ in $G$ we have $d_G(x_{j+1}, s) = d_G(x_j, s) + 1$. Therefore we get $d_{G_0}(x_{j+1}, s) \leq d_G(x_{j+1}, s) + S_j\delta$. Since $S_{j+1} = S_j$, the desired inequality follows. □

**Lemma 3 (Restated).** *If an edge $(u,v)$ is inserted such that $d_{G_0}(u,s) > d_{G_0}(v,s) + \delta$, then*

$$d_{G_0}(u,s) \geq d_G(u,s) + \delta\,.$$

*Proof.* If this situation occurs in the algorithm, then it is because we inserted an edge $(u,v)$ such that $d_{G_0}(u,s) > d_{G_0}(v,s)+\delta$ (which is equivalent to $d_{G_0}(v,s) \leq d_{G_0}(u,s)-\delta-1$). In the current graph $G$, we already have inserted the edge $(u,v)$

and therefore $d_G(u,s) \leq d_G(v,s)+1$. Since $G$ is the result of inserting edges into $G_0$, distances in $G$ are not longer than in $G_0$, and in particular $d_G(v,s) \leq d_{G_0}(v,s)$. Therefore we arrive at the following chain of inequalities:

$$d_G(u,s) \leq d_G(v,s)+1 \leq d_{G_0}(v,s)+1 \leq d_{G_0}(u,s)-\delta-1+1 = d_{G_0}(u,s)-\delta$$

Thus, we get $d_{G_0}(u,s) \geq d_G(u,s)+\delta$. □

**Lemma 4 (Restated).** *Let $G=(V,E)$ and $G'=(V,E')$ be unweighted, undirected graphs on the same set of nodes $V$ such that $G$ is connected and $E(G) \subseteq E(G')$. For every node $s$, if there is a node $y$ such that $d_G(y,s) \geq d_{G'}(y,s)+\delta$, then*

$$\sum_{x \in V} d_G(x,s) \geq \sum_{x \in V} d_{G'}(x,s) + \Omega(\delta^2)\,.$$

*Proof.* Let $P$ denote the shortest path from $y$ to $s$ of length $d_G(y,s)$ in $G$. We first bound the distance change for single nodes.

*Claim.* For every node $x$ on $P$ we have $d_G(x,s) \geq d_{G'}(x,s)+\delta-2d_G(x,y)$

*Proof.* Suppose that there is some node $x$ on $P$ such that $d_G(x,s) < d_{G'}(x,s)+\delta-2d_G(x,y)$. Since $x$ is on the shortest path $P$ we have $d_G(y,s) = d_G(y,x)+d_G(x,s)$. We also need the inequality $d_{G'}(x,s)-d_G(y,x) \leq d_{G'}(y,s)$ which we get as follows:

$$d_{G'}(x,s) \leq \underbrace{d_{G'}(x,y)}_{\leq d_G(x,y)} + d_{G'}(y,s) \leq d_G(x,y)+d_{G'}(y,s)$$

Putting this together we get:

$$\begin{aligned} d_G(y,s) = d_G(y,x) + \underbrace{d_G(x,s)}_{<d_{G'}(x,s)+\delta-2d_G(x,y)} &< d_G(y,x)+d_{G'}(x,s)+\delta-2d_G(x,y) \\ &= \underbrace{d_{G'}(x,s)-d_G(y,x)}_{\leq d_{G'}(y,s)} + \delta \leq d_{G'}(y,s)+\delta \end{aligned}$$

However, this contradicts the assumption that $d_G(y,s) \geq d_{G'}(y,s)+\delta$. □

From the claim we conclude that

$$\begin{aligned}
\sum_{x\in P, d_G(x,y)<\delta/2} d_G(x,s) &\geq \sum_{x\in P, d_G(x,y)<\delta/2} (d_{G'}(x,s) + \delta - 2d_G(x,y)) \\
&= \sum_{x\in P, d_G(x,y)<\delta/2} d_{G'}(x,s) + \sum_{x\in P, d_G(x,y)<\delta/2} (\delta - 2d_G(x,y)) \\
&\geq \sum_{x\in P, d_G(x,y)<\delta/2} d_{G'}(x,s) + \sum_{j=1}^{\lfloor \delta/2 \rfloor} (\delta - 2j) \\
&= \sum_{x\in P, d_G(x,y)<\delta/2} d_{G'}(x,s) + \sum_{j=1}^{\lfloor \delta/2 \rfloor} \delta - 2\sum_{j=1}^{\lfloor \delta/2 \rfloor} j \\
&= \sum_{x\in P, d_G(x,y)<\delta/2} d_{G'}(x,s) + \delta(\lfloor \delta/2 \rfloor) - \lfloor \delta/2 \rfloor(\lfloor \delta/2 \rfloor + 1) \\
&= \sum_{x\in P, d_G(x,y)<\delta/2} d_{G'}(x,s) + \Omega(\delta^2)
\end{aligned}$$

Finally, we get:

$$\begin{aligned}
\sum_{x\in V} d_G(x,s) &= \sum_{x\in P, d_G(x,y)<\delta/2} d_G(x,s) + \sum_{x\notin P \text{ or } d_G(x,y)\geq\delta/2} \underbrace{d_G(x,s)}_{\geq d_{G'}(x,s)} \\
&\geq \sum_{x\in P, d_G(x,y)<\delta/2} d_{G'}(x,s) + \Omega(\delta^2) + \sum_{x\notin P \text{ or } d_G(x,y)\geq\delta/2} d_{G'}(x,s) \\
&= \sum_{x\in V} d_{G'}(x,s) + \Omega(\delta^2)
\end{aligned}$$

□

**Lemma 5 (Restated).** *If $\kappa \leq q$ and $\delta \leq X$, then the total running time of Algorithm 1 is $O(T_{BFS}(X) \cdot q/\kappa + T_{BFS}(X) \cdot nX/\delta^2 + q)$ where $T_{BFS}(X)$ is the time needed for computing a BFS tree up to depth $X$.*

*Proof.* Besides the constant time per deletion we have to compute a BFS tree of depth $X$ at the beginning of every phase. The first cause for starting a new phase is that the number of edge deletions in a phase reaches $\kappa$, which can happen at most $q/\kappa$ times. The second cause for starting a new phase is that we insert an edge $(u, v)$ such that $d_{G_0}(u, s) > d_{G_0}(v, s) + \delta$. By Lemmas 3 and 4 this implies that the sum of the distances of all nodes to $s$ has increased by at least $\Omega(\delta^2)$ since the beginning of the current phase. There are at most $n$ nodes of distance at most $X$ to $s$ which means that the sum of the distances is at most $nX$. Therefore such a decrease can occur at most $O(nX/\delta^2)$ times. □

**Theorem 6 (Restated).** *In the RAM model, there is an incremental $(1+\epsilon)$-approximate SSSP algorithm for inserting up to $q$ edges that has a total update time of $O(mn^{2/5}q^{2/5}/\epsilon)$ where $m$ is the number of edges in the final graph. It answers distance and path queries in optimal worst-case time.*

*Proof.* In parallel to Algorithm 1 (for which we set $X = n$), we keep an Even-Shiloach tree with root $s$ up to depth $\kappa\delta/\epsilon$ which maintains the *exact* distance of every node $x$ such that $d_G(x,s) \leq \kappa\delta/\epsilon$. Thus, Algorithm 1 only needs to provide a $(1+\epsilon)$-approximation for every node $v$ such that $d_G(x,s) \geq \kappa\delta/\epsilon$. This is indeed the case, as can be seen by applying Lemma 2

$$d_{G_0}(x,s) \leq d_G(x,s) + \kappa\delta \leq d_G(x,s) + \epsilon d_G(x,s) = (1+\epsilon)d_G(x,s)\,.$$

Maintaining the Even-Shiloach tree takes time $O(m\kappa\delta/\epsilon)$ (see Theorem 14). The time for computing a BFS tree from scratch is $O(m)$. By Lemma 5 (using $X = n$) the total running time for the combination of both algorithms is $O(m(\kappa\delta/\epsilon + q/\kappa + n^2/\delta^2))$. We can balance the three terms by setting $\kappa = q^{3/5}/n^{2/5}$ and $\delta = n^{4/5}/q^{1/5}$ which gives the total running time $O(mn^{2/5}q^{2/5}/\epsilon)$. Note that Lemma 5 demands that $\kappa \leq q$ and $\delta \leq X = n$. It is easy to verify that our choice of $\kappa$ and $\delta$ fulfills this requirement.

The final number $q$ of insertions does not have to be known beforehand. We use a doubling approach for for guessing the value of $q$ where the $i$-th guess is $q_i = 2^i$. When the number of insertions exceeds our guess $q_i$, we simply start a new phase and use the guess $q_{i+1} = 2q_i$ from now on. The total running time for this approach is $O(\sum_{i=1}^{\log q} mn^{2/5}q_i^{2/5}/\epsilon)$ which is $O(mn^{2/5}q^{2/5}/\epsilon)$ because

$$\sum_{i=1}^{\log q}(2^i)^{2/5} = \sum_{i=1}^{\log q}(2^{2/5})^i = O(q)\,.$$

□

**Lemma 15.** *If $\kappa \leq q$ and $\delta \leq X$, then the total running time of Algorithm 1 in the distributed model is*

$$O\left(\frac{qX}{\kappa} + \frac{nX^2}{\delta^2} + q\right).$$

*Proof.* Computing a BFS tree up to depth $X$ takes time $O(X)$ in the distributed model. Therefore the desired running time directly follows from Lemma 5.

In the distributed model, we have to take care of some implementation details of the algorithm because not all information is globally available to every node. For every insertion of an edge $(u,v)$, the nodes $u$ and $v$ can exchange their initial distances $d_{G_0}(u,s)$ and $d_{G_0}(v,s)$ in constant time.

The root node also needs some special information to coordinate the phases. The first cause for starting a new phase is when the level of some node decreases by at least $\delta$. If a node detects a level decrease by at least $\delta$, it has to inform the root $s$ about the decrease so that $s$ can initiate the beginning of the next phase. The tree maintained by our algorithm, which has depth at most $X$, can be used

to send this message. Therefore the total time needed for send this message is $O(X)$. This running time corresponds to the construction of the new BFS tree and therefore can be charged to the new phase.

The second cause for starting a new phase is that the number of edge insertions since the beginning of the current phase exceeds $\kappa$. Therefore it is necessary that the root $s$ knows the number of edges that have been inserted. After the insertion of an edge $(u, v)$ the node $v$ sends a message to $s$ to inform about the edge insertion. Note that messages of this type do not have to arrive at the root node $s$ immediately and therefore they do not have to be sent exclusively during recovery phases. Again, the tree maintained by our algorithm can be used to send these messages.

**Lemma 16.** *By setting*

$$\kappa = \frac{q^{1/3} X^{1/3} \gamma^{2/3}}{n^{1/3}}$$

*and*

$$\delta = \frac{n^{1/3} X^{2/3} \gamma^{1/3}}{q^{1/3}}$$

*(where $\gamma = \epsilon/4$), Algorithm 1 has a total running time of*

$$O\left(\frac{q^{2/3} n^{1/3} X^{2/3}}{\epsilon^{2/3}} + q\right)$$

*provided that $X \geq n/q$. Furthermore, it has the following invariant: For every node $x$ such that $X/2 \leq d_{G_0}(x, s) \leq X$ we have*

$$d_G(x, s) \leq d_T(x, s) \leq d_{G_0}(x, s) \leq (1 + \epsilon) d_G(x, s)\,.$$

*Proof.* To simplify the notation a bit we define $A = \kappa\delta$, which gives $A = \gamma X \leq X$. First, we have to make sure that our choices of $\kappa$ and $\delta$ are actually valid, i.e., we have to check whether $\kappa \leq q$ and $\delta \leq X$. Since $X \leq n$ we get

$$\kappa = \frac{q^{1/3} X^{1/3} \gamma^{2/3}}{n^{1/3}} \leq \frac{q^{1/3} n^{1/3} \gamma^{2/3}}{n^{1/3}} \leq q^{1/3} \leq q\,.$$

We have also assumed that $X \geq n/q$ which is equivalent to $n \leq qX$ and therefore we get

$$\delta = \frac{n^{1/3} X^{2/3} \gamma^{1/3}}{q^{1/3}} \leq \frac{q^{1/3} X^{1/3} X^{2/3} \gamma^{1/3}}{q^{1/3}} = \gamma^{1/3} X \leq X$$

By Lemma 15, Algorithm 1 runs in time

$$O\left(X\frac{q}{\kappa} + X\frac{nX}{\delta^2} + q\right)$$

It is easy to check that by our choices of $\kappa$ and $\delta$ the two terms appearing in the running time are balanced and we get

$$\frac{Xq}{\kappa} = \frac{nX^2}{\delta^2} = \frac{q^{2/3} n^{1/3} X^{2/3}}{\gamma^{2/3}} = O\left(\frac{q^{2/3} n^{1/3} X^{2/3}}{\epsilon^{2/3}}\right)$$

We now argue that the invariant holds. By Lemma 2, we already know that

$$d_G(x,s) \leq d_T(x,s) \leq d_{G_0}(x,s) \leq d_G(x,s) + A$$

for every node $x$ such that $d_{G_0}(x,s) \leq X$. We now show that our choice of $\kappa$ and $\delta$ guarantees that $A \leq \epsilon d_G(x,s)$, for every node $x$ such that $d_{G_0}(x,s) \geq X/2$, which immediately gives the desired inequality.

Assume that $d_{G_0}(x,s) \leq d_G(x,s) + A$ and that $d_{G_0}(x,s) \geq X/2$. We first show that

$$\gamma \leq \frac{1}{2(1+\frac{1}{\epsilon})}\,.$$

Since $\epsilon \leq 1$ we have $2(\epsilon+1) \leq 4$. It follows that

$$\frac{1}{2(\epsilon+1)} \geq \frac{1}{4}$$

which is equivalent to

$$\frac{1}{2(1+\frac{1}{\epsilon})} \geq \frac{\epsilon}{4} = \gamma$$

Therefore we get the following chain of inequalities:

$$\left(1+\frac{1}{\epsilon}\right)A = \left(1+\frac{1}{\epsilon}\right)\gamma X \leq \frac{\left(1+\frac{1}{\epsilon}\right)X}{2(1+\frac{1}{\epsilon})} = \frac{X}{2} \leq d_{G_0}(x,s)$$

We now subtract $A$ from both sides and get

$$\frac{A}{\epsilon} \leq d_{G_0}(x,s) - A\,.$$

Since $d_{G_0}(x,s) - A \leq d_G(x,s)$ by assumption, we finally get $A \leq \epsilon d_G(x,s)$. □

**Theorem 7 (Restated).** *In the distributed model, there is an incremental algorithm for maintaining a $(1+\epsilon)$-approximate BFS tree under up to $q$ edge insertions that has a total running time of*

$$O((q^{2/3}n^{1/3}D^{2/3}/\epsilon^{2/3} + q)\log D + n)$$

*where $D$ is the initial diameter.*

*Proof.* We run $\log D$ parallel instances of Algorithm 1 where, for the $i$-th instance we set the parameter $X$ to $X_i = 2^i$ and $\kappa$ and $\delta$ as in Lemma 16. If we detect at the initialization of some instance $i$ that the initial BFS tree already contains all nodes, we will not start any instance for a larger value of $X$. Therefore the maximal value we use for $X$ will be $O(D)$. Every time a phase ends for instance $i$, we also start a new phase for every instance $j$ such that $j \leq i$. This guarantees that if a node falls out of the range $[X_i/2, X_i]$ it will immediately be covered by

a lower range. By using the bound $X_i \leq D$, the running time of each instance of Algorithm 1 is $O(q^{2/3}n^{1/3}D^{2/3}/\epsilon^{2/3} + q)$.

Note that Lemma 16 does not apply if $X < n/q$. If $q \leq n$, we simply recompute the BFS tree of depth $n/q$ after every edge deletion. This adds $O(qn/q) = O(n)$ to the total running time. If $q > n$, we maintain, also in parallel, an Even-Shiloach tree of depth $n/q$ from the root $s$. This can be done in time $O(n^2/q)$ which is $O(q)$ since $n \leq q$. Thus, the total running time of this algorithm is $O((q^{2/3}n^{1/3}D^{2/3}/\epsilon^{2/3} + q)\log D + n)$.

By using a doubling approach for guessing the value of $q$ we can run the algorithm without knowing the number of edge insertions beforehand. □

## B Omitted Proofs of Section 4

**Lemma 8 (Restated).** *After every iteration of the tree update procedure of Algorithm 2, we have, for all nodes $x$ and $y$ such that $d^w_F(x,y) < \infty$,*

$$d^w_F(x,y) + d_{G_0}(y,s) \leq d_{G_0}(x,s) + \kappa\delta\,.$$

*Proof.* By $F_i$ and $U_i$ we denote the graph $F$ and the set $U$ after the $i$-th iteration of the algorithm, respectively. We call the weighted edges inserted by the tree update procedure *artificial edges*. In the graph $F_i$ there are two types of edges: those that were already present in the BFS tree $T_0$ and artificial edges. For every edge deletion we add exactly one artificial edge. Therefore there are at most $\kappa$ artificial edges in $F_i$.

Consider the shortest path $P = x_l, x_{l-1}, \ldots x_0$ from $x$ to $y$, where $x_l = x$ and $x_0 = y$. Let $S_j$ (with $0 \leq j \leq l$) denote the number of artificial edges in the subpath $x_j, x_{j-1}, \ldots x_0$. Now consider the following claim.

*Claim.* For every integer $j$ with $0 \leq j \leq l$ we have

$$d^w_{F_i}(x_j,y) + d_{G_0}(y,s) \leq d_{G_0}(x_j,s) + S_j\delta\,.$$

Assuming the truth of the claim, the desired inequality follows straightforwardly. The claim gives $d^w_{F_i}(x_l,y)+d_{G_0}(y,s) \leq d_{G_0}(x_l,s)+S_l\cdot\delta$ Since $S_l \leq \kappa$ and $x_l = x$, we get that $d^w_{F_i}(x,y) + d_{G_0}(y,s) \leq d_{G_0}(x,s) + \kappa\cdot\delta$.

In the following we prove the claim by induction on $j$ We first consider the induction base $j = 0$ for which we have $x_j = y$ and $S_j = 0$. The inequality then trivially holds due to $d^w_{F_i}(y,y) = 0$. We now prove the inductive step from $j$ to $j+1$. Note that $S_j \leq S_{j+1} \leq S_j + 1$ since the path is exactly one edge longer. Consider first the case that $(x_{j+1}, x_j)$ is an edge from the BFS tree $T_0$ of the graph $G_0$. In that case we have $d^w_{F_i}(x_{j+1},x_j) = d_{G_0}(x_{j+1},x_j) = 1$. Furthermore, since $(x_{j+1}, x_j)$ is an edge in the BFS tree $T_0$ we know that $x_j$ lies on a shortest path from $x_{j+1}$ to $s$ in $G_0$. Therefore we have $d_{G_0}(x_{j+1},s) =$

$d_{G_0}(x_{j+1}, x_j) + d_{G_0}(x_j, s)$. Together with the induction hypothesis we get:

$$\begin{aligned}
\underbrace{d^w_{F_i}(x_{j+1}, y)}_{=d^w_{F_i}(x_{j+1},x_j)+d^w_{F_i}(x_j,y)} + d_{G_0}(y, s) &= \underbrace{d^w_{F_i}(x_{j+1}, x_j)}_{=d_{G_0}(x_{j+1},x_j)} + \underbrace{d^w_{F_i}(x_j, y) + d_{G_0}(y, s)}_{\leq d_{G_0}(x_j,s)+S_j\cdot\delta \text{ (by IH)}} \\
&\leq \underbrace{d_{G_0}(x_{j+1}, x_j) + d_{G_0}(x_j, s)}_{=d_{G_0}(x_{j+1},s)} + \underbrace{S_j}_{=S_{j+1}} \cdot\delta \\
&= d_{G_0}(x_{j+1}, s) + S_{j+1} \cdot \delta\,.
\end{aligned}$$

The second case is that $(x_{j+1}, x_j)$ is an artificial edge. In that case we have $d^w_{F_i}(x_{j+1}, x_j) = d_G(x_{j+1}, x_j)$ and by the algorithm the inequality $d_G(x_{j+1}, x_j) + d_{G_0}(x_j, s) + \leq d_{G_0}(x_{j+1}, s) + \delta$ holds. Note also that $S_{j+1} = S_j + 1$. We therefore get the following:

$$\begin{aligned}
\underbrace{d^w_{F_i}(x_{j+1}, y)}_{=d^w_{F_i}(x_{j+1},x_j)+d^w_{F_i}(x_j,y)} + d_{G_0}(y, s) &= \underbrace{d^w_{F_i}(x_{j+1}, x_j)}_{=d_G(x_{j+1},x_j)} + \underbrace{d^w_{F_i}(x_j, y) + d_{G_0}(y, s)}_{\leq d_{G_0}(x_j,s)+S_j\cdot\delta \text{ (by IH)}} \\
&\leq \underbrace{d_G(x_{j+1}, x_j) + d_{G_0}(x_j, s)}_{\leq d_{G_0}(x_{j+1},s)+\delta} + S_j \cdot \delta \\
&= d_{G_0}(x_{j+1}, s) + \underbrace{(S_j + 1)}_{=S_{j+1}} \cdot\delta \\
&= d_{G_0}(x_{j+1}, s) + S_{j+1} \cdot \delta
\end{aligned}$$

□

**Lemma 17.** *After every iteration of the tree update procedure of Algorithm 2, we have, for all nodes $x$ and $y$ such that $d^w_{F_i}(x, y) < \infty$,*

$$d_{G_0}(y, s) - d_{G_0}(x, s) \leq \kappa\delta\,.$$

*Proof.* By $F_i$ and $U_i$ we denote the graph $F$ and the set $U$ after the $i$-th iteration of the algorithm, respectively. We distinguish two cases. The first case is that $d_{G_0}(y, s) \leq d_{G_0}(x, s)$ which trivially gives $d_{G_0}(y, s) - d_{G_0}(x, s) \leq 0 \leq \kappa\delta$. The second case is $d_{G_0}(y, s) > d_{G_0}(x, s)$ for which we get

$$d_{G_0}(x, s) + \kappa\delta \underbrace{\geq}_{\text{Lemma 8}} \underbrace{d^w_{F_i}(x, y)}_{\geq d_{G_0}(x,y)} + \underbrace{d_{G_0}(y, s)}_{\geq d_{G_0}(x,s)} \geq \underbrace{d_{G_0}(x, y)}_{=d_{G_0}(y,x)} + d_{G_0}(x, s) \geq d_{G_0}(y, s)$$

which is equivalent to $d_{G_0}(y, s) - d_{G_0}(x, s) \leq \kappa\delta$. □

**Lemma 9 (Restated).** *After every iteration of the tree update procedure of Algorithm 2, the graph $F$ is a forest.*

*Proof.* The proof will use Lemma 17 which we prove directly above. By $F_i$ and $U_i$ we denote the graph $F$ and the set $U$ after the $i$-th iteration of the algorithm, respectively. The claim is definitely true at the beginning of the algorithm because $F_0$ is the result of deleting edges from a tree. Now assume that $F_i$ is a forest and $F_{i+1}$ is not a forest. We can partition the forest $F_i$ into the following components: for every node $u \in U_i \cup \{s\}$ there is a component that contains every node $x$ for which there is a directed path from $x$ to $u$. Therefore the only ways of introducing a cycle are the following:

- The algorithm adds an edge $(u_1, v_l)$ such that there is a path from $v_l$ to $u_1 \in U_i$ in $F_i$ and $v_1$ is not marked in iteration $i$.
- The algorithm adds some edges $(u_1, v_1), \ldots, (u_l, v_l)$ (where $l \geq 2$) such that:
  - there is a path from $v_1$ to $u_2 \in U_i$ in $F_i$ and $v_1$ is not marked in iteration $i$
  - there is a path from $v_2$ to $u_3 \in U_i$ in $F_i$ and $v_2$ is not marked in iteration $i$
  - $\ldots$
  - there is a path from $v_l$ to $u_1 \in U_i$ in $F_i$ and $v_l$ is not marked in iteration $i$

For any of these situations, there is a path from $u_1$ to $v_l$ in $F_{i+1}$. Therefore we may apply Lemma 8 and get

$$d^w_{F_{i+1}}(u_1, v_l) + d_{G_0}(v_l, s) \leq d_{G_0}(u_1, s) + \kappa\delta$$

which is equivalent to

$$d^w_{F_{i+1}}(u_1, v_l) \leq d_{G_0}(u_1, s) - d_{G_0}(v_l, s) + \kappa\delta\,.$$

Since there is a path from $v_l$ to $u_1$ in $F_i$ we may apply Lemma 17 and get $d_{G_0}(u_1, s) - d_{G_0}(v_l, s) \leq \kappa\delta$. Therefore we have $d^w_{F_{i+1}}(u_1, v_l) \leq 2\kappa\delta$. The combination of both inequalities gives $d^w_{F_{i+1}}(u_1, v_l) \leq 2\kappa\delta$.

Note that $d_{G_0}(v_l, u_1) = d_{G_0}(u_1, v_l) \leq d^w_{F_{i+1}}(u_1, v_l) \leq 2\kappa\delta$ and thus $d_{G_0}(v_l, u_1) \leq 2\kappa\delta$. By the triangle inequality we get

$$d_{G_0}(v_l, s) \leq \underbrace{d_{G_0}(v_l, u_1)}_{\leq 2\kappa\delta} + d_{G_0}(u_1, s) \leq d_{G_0}(u_1, s) + 2\kappa\delta\,.$$

We now apply Lemma 8 again and get

$$d^w_{F_i}(v_l, u_1) + d_{G_0}(u_1, s) \leq \underbrace{d_{G_0}(v_l, s)}_{\leq d_{G_0}(u_1, s) + 2\kappa\delta} + \kappa\delta \leq d_{G_0}(u_1, s) + 3\kappa\delta$$

which is equivalent to $d^w_{F_i}(v_l, u_1) \leq 3\kappa\delta$. But then, since $u_1 \in U_i$, $v_l$ must be marked which is a contradiction to the fact that $v_l$ is not marked. Therefore our assumption was wrong and we conclude that $F_{i+1}$ is a forest. □

**Lemma 18.** *Let $u$ be a node such that $d_G(u, s) \leq d_{G_0}(u, s) + \delta$. For every node $x$ on a shortest path from $u$ to $s$ in $G$ we have*

$$d_G(u, x) + d_{G_0}(x, s) \leq d_{G_0}(u, s) + \delta\,.$$

*Proof.* Since $x$ on a shortest path from $u$ to $s$ in $G$ we have $d_G(u,s) = d_G(u,x) + d_G(x,s)$. This gives:

$$d_G(u,x) + \underbrace{d_{G_0}(x,s)}_{\leq d_G(x,s)} \leq d_G(u,x) + d_G(x,s) = d_G(u,s) \leq d_{G_0}(u,s) + \delta\,.$$

□

**Lemma 10 (Restated).** *In every iteration of the tree update procedure of Algorithm 2, if $d_G(x,s) \leq d_{G_0}(x,s) + \delta$ for every node $x \in U$, then for every node $u \in U$ with minimal $d_{G_0}(u,s)$, there is a node $v \in V$ such that*

*(1)* $d_G(u,v) + d_{G_0}(v,s) \leq d_{G_0}(u,s) + \delta$
*(2)* $v$ *is not marked*
*(3)* $d_G(u,v) \leq (\kappa+1)\delta + 1$

*Proof.* The proof will use Lemmas 17 and 18 which we prove above. By $F_i$ and $U_i$ we denote the graph $F$ and the set $U$ after the $i$-th iteration of the algorithm, respectively. Let $u \in U_i$ denote a node that has minimal distance to $s$ in $G_0$ among all nodes in $U$, i.e. $d_{G_0}(u,s) \leq d_{G_0}(x,s)$ for all $x \in U_i$. We now argue about $F_i$ after the algorithm has done the marking. Note that $s$ does not have a parent in $F_i$. Therefore $s$ is not marked. Now consider a shortest path $P$ from $u$ to $s$ in $G$. This shortest path contains at least one node that is marked, namely $u$, and at least one node that is not marked, namely $s$. We denote by $v$ first node that is not marked if we go along $P$ from $u$ to $s$. By $x$ we denote the predecessor of $v$ on the path $P$. Note that by this definition $x$ is marked.

We now argue that $v$ fulfills all the properties we desire. Property (2) holds by the way we defined $v$ and property (1) follows by Lemma 18 because $v$ lies on a shortest path from $u$ to $s$.

Since $x$ is on a shortest path from $u$ to $s$ in $G$ we have, by Lemma 18,

$$d_G(u,x) + d_{G_0}(x,s) \leq d_{G_0}(u,s) + \delta\,. \tag{1}$$

Since $x$ is marked there is some $u' \in U_i$ such that $x$ is in the subtree of $u'$ in $F_i$. By our definition of $u$ we have $d_{G_0}(u,s) \leq d_{G_0}(u',s)$. We now rewrite inequality (1) and get

$$d_G(u,x) - \delta \leq \underbrace{d_{G_0}(u,s)}_{\leq d_{G_0}(u',s)} - d_{G_0}(x,s) \leq d_{G_0}(u',s) - d_{G_0}(x,s) \underbrace{\leq}_{\text{Lemma 17}} \kappa\delta\,.$$

By adding $\delta$ on both sides we get $d_G(x,u) \leq (\kappa+1)\delta$.

Since $x$ and $v$ are neighbors on the path $P$ in $G$ we know that $G$ contains the edge $(x,v)$. Therefore we get

$$d_G(u,v) \leq \underbrace{d_G(u,x)}_{\leq(\kappa+1)\delta} + \underbrace{d_G(x,v)}_{=1} \leq (\kappa+1)\delta + 1$$

□

**Lemma 11 (Restated).** *The tree update procedure of Algorithm 2 either reports "distance increase" and guarantees that there is a node $x$ such that*

$$d_G(x,s) > d_{G_0}(x,s) + \delta$$

*or it computes a tree $T$ such that for every node $x$ we have*

$$d_{G_0}(x,s) \leq d_G(x,s) \leq d_T^w(x,s) \leq d_{G_0}(x,s) + \kappa\delta\,.$$

*It runs in time $O(\kappa^2\delta)$.*

*Proof.* Consider first the case that the algorithm reports "distance increase", i.e., in some iteration $i$, for every node $u \in U_i$ there is no node $v$ such that

(1) $d_G(u,v) + d_{G_0}(v,s) \leq d_{G_0}(u,s) + \delta$,
(2) $v$ is not marked, and
(3) $d_G(u,v) \leq (\kappa+1)\delta + 1$.

From Lemma 10 it follows (by modus tollens) that there is a node $y$ such that $d_G(y,s) > d_{G_0}(y,s) + \delta$.

Consider now the case that the algorithm does not report "distance increase". By Lemma 9 the graph $T$ computed by the algorithm is a forest. Since the algorithm is finished, every node in this forest has an outgoing edge, which makes it a tree. It is clear that $d_{G_0}(x,s) \leq d_G(x,s)$ because $G$ results from $G_0$ by some edge deletions.

We now prove the inequality $d_T^w(x,s) \leq d_{G_0}(x,s) + \kappa\delta$. Since $T$ is a tree with root $s$ containing all nodes, there is a path from $x$ to $s$ in $T$. Therefore we may apply Lemma 8 and get

$$d_T^w(x,s) + d_{G_0}(s,s) \leq d_{G_0}(x,s) + \kappa\delta\,.$$

The desired inequality follows since $d_{G_0}(s,s) = 0$.

We now prove the inequality $d_G(x,s) \leq d_T^w(x,s)$. Consider the unique path going from $x$ to $s$ in $T$ consisting of the nodes $x = x_l, x_{l-1}, \ldots x_0 = s$. We know that every edge $(x_{j+1}, x_j)$ in $T$ either was part of the initial BFS tree $T_0$, which means that $d_T^w(x_{j+1},x_j) = 1 = d_{G_0}(x_{j+1},x_j)$, or was inserted later by the algorithm, which means that $d_T^w(x_{j+1},x_j) = d_{G_0}(x_{j+1},x_j)$. This means that in any case we have $d_T^w(x_{j+1},x_j) = d_{G_0}(x_{j+1},x_j)$ and therefore we get

$$d_T^w(x,s) = \sum_{j=0}^{l-1} \underbrace{d_T^w(x_{j+1},x_j)}_{=d_G(x_{j+1},x_j)} \geq \sum_{j=0}^{l-1} d_G(x_{j+1},x_j) \geq d_G(x,s)\,.$$

Finally, we argue about the running time. The algorithm has two parts: the marking process and the search process. In the marking process, every node in $U$ marks its subtree in $F$ up to distance $2\kappa\delta$. For a single node this needs time $O(\kappa\delta)$. Therefore the running time of the reconnection part is $O(\kappa\delta)$ and $U$

decreases with every iteration. In the search process, every node $u \in U$ wants to find a node $v$ to connect to that fulfills certain properties. We search for such a node $v$ by examining the neighborhood of $u$ in $G$ up to depth $(\kappa+1)\delta+1$ using a BFS strategy. Both parts of our algorithm run in time $O(\kappa\delta)$ per iteration. There are at most $|U|$ iterations and the size of $U$ is at most $\kappa$. □

**Lemma 19.** *If $X \geq 2^{7/2}n/(q\epsilon^{7/2})$, then, by setting $\kappa = q^{1/7}X^{1/7}/n^{1/7}$ and $\delta = n^{3/7}X^{4/7}/q^{3/7}$, Algorithm 2 runs in time*

$$O\left(q^{6/7}n^{1/7}X^{6/7}\right).$$

*Furthermore, it has the following invariant: For every node $x$ such that $d_{G_0}(x,s) \leq X$ and $d_G(x,s) \geq X/2$ we have*

$$d_{G_0}(x,s) \leq d_G(x,s) \leq d_T^w(x,s) \leq (1+\epsilon)d_{G_0}(x,s).$$

*Proof.* It is easy to check that by our choices of $\kappa$ and $\delta$ the three terms in the running time of Lemma 12 are balanced and we get:

$$\frac{r}{\kappa} \cdot X = \frac{nX}{\delta^2} \cdot X = q\kappa^2\delta = q^{6/7}n^{1/7}X^{6/7}.$$

We also have to check that $\kappa \leq q$ and $\delta \leq X$. Since $X \leq n$ we get

$$\kappa = \frac{q^{1/7}X^{1/7}}{n^{1/7}} \leq \frac{q^{1/7}n^{1/7}}{n^{1/7}} = q^{1/7} \leq q.$$

By assumption we have $X \geq 2^{7/2}n/(q\epsilon^{7/2})$. Since $\epsilon \leq 1$ it follows that $X \geq n/q$, which is equivalent to $n \leq qX$ and therefore we get

$$\delta = \frac{n^{3/7}X^{4/7}}{q^{3/7}} = \frac{q^{3/7}X^{3/7}X^{4/7}}{q^{3/7}} = X.$$

We now argue that the invariant holds. By Lemma 12, we already know that

$$d_{G_0}(x,s) \leq d_G(x,s) \leq d_T^w(x,s) \leq d_{G_0}(x,s) + \kappa\delta$$

for every node $x$ such that $d_{G_0}(x,s) \leq X$. We now show that our choice of $\kappa$ and $\delta$ guarantees that $\kappa\delta \leq \epsilon d_{G_0}(x,s)$, for every node $x$ such that $d_{G_0}(x,s) \geq X/2$, which immediately gives the desired inequality. By assumption we have $X \geq 2^{7/2}n/(q\epsilon^{7/2})$, which is equivalent to $n \leq \epsilon^{7/2}qX/2^{7/2}$ and therefore we get

$$\kappa\delta = \frac{q^{1/7}X^{1/7}}{n^{1/7}} \cdot \frac{n^{3/7}X^{4/7}}{q^{3/7}} = \frac{n^{2/7}X^{5/7}}{q^{2/7}} \leq \frac{\epsilon q^{2/7}X^{2/7}X^{5/7}}{2q^{2/7}} = \frac{\epsilon X}{2} \leq \epsilon d_{G_0}(x,s).$$

□

**Theorem 13 (Restated).** *In the decremental distributed dynamic network, we can maintain a $(1+\epsilon)$-approximate BFS tree over $q$ edge deletions in a total running time of*

$$O\left(\left(q^{6/7}n^{1/7}D^{6/7}\right)\log D + \frac{n+q}{\epsilon^{7/2}}\right),$$

*where $D$ is the maximum diameter.*

*Proof.* The algorithm works as follows. At any time, we run $\log D$ parallel instances of Algorithm 2 where, for the $i$-th instance we set the parameter $X$ to $X_i = 2^i$ and $\kappa$ and $\delta$ as in Lemma 19. Instance $i$ covers the range $[X_i/2, X]$ Note that $D$ might increase due to edge deletions. When we start a new phase in the instance that currently covers the highest range $[X_i/2, X]$, we do a full BFS tree computation. If the depth of the BFS tree exceeds $X$, we start new instances covering the new range and charge the running time of the BFS tree computation to them. By using the bound $X_i \leq D$, the running time of each instance of Algorithm 1 is

$$O\left(q^{6/7}n^{1/7}D^{6/7}\right)$$

Note that Lemma 19 does not apply if $X < 2^{7/2}n/(q\epsilon^{7/2})$. If $q \leq n$, we simply recompute the BFS tree of depth $2^{7/2}n/(q\epsilon^{7/2})$ after every edge deletion. This adds $O(qn/(q\epsilon^{7/2})) = O(n/\epsilon^{7/2})$ to the total running time. If $q > n$, we maintain, also in parallel, an Even-Shiloach tree of depth $2^{7/2}n/(q\epsilon^{7/2})$ from the root $s$. This can be done in time $O(n^2/(q\epsilon^{7/2})+q)$ which is $O(q/\epsilon^{7/2})$ since $n \leq q$ and $\epsilon \leq 1$. Thus, the total running time of this algorithm is

$$O\left(q^{6/7}n^{1/7}D^{6/7}\log D + \frac{n+q}{\epsilon^{7/2}}\right)$$

By using a doubling approach for guessing the value of $q$ we can run the algorithm without knowing the number of edge deletions $q$ beforehand. □