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

MONIKA HENZINGER, SEBASTIAN KRINNINGER, and DANUPON NANONGKAI,
University of Vienna, Faculty of Computer Science

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 present deterministic $(1 + \epsilon)$-approximation algorithms whose amortized time (over some number of link changes) is *sublinear* in D, the *maximum diameter* of the network.

Our technique also leads to a deterministic $(1 + \epsilon)$-approximate incremental algorithm for single-source shortest paths in the sequential (usual RAM) model. Prior to our work, the state of the art was the classic *exact* algorithm of Even and Shiloach (1981), which is optimal under some assumptions (Roditty and Zwick 2011; Henzinger et al. 2015). Our result is the first to show that, in the incremental setting, this bound can be beaten in certain cases if some approximation is allowed.

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms; Network problems*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Distributed algorithms, dynamic algorithms

ACM Reference format:
Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2017. Sublinear-Time Maintenance of Breadth-First Spanning Trees in Partially Dynamic Networks. *ACM Trans. Algorithms* 13, 4, Article 51 (December 2017), 24 pages.
https://doi.org/10.1145/3146550

A preliminary version of this article was presented at the International Colloquium on Automata, Languages and Programming (ICALP) 2013 (Henzinger et al. 2013). The research leading to this work has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 317532 and from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant Agreement no. 340506. S. Krinninger was supported in part by the University of Vienna (IK I049-N). D. Nanongkai was supported in part by the following research grants: Nanyang Technological University grant M58110000, Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 2 grant MOE2010-T2-2-082, and Singapore MOE AcRF Tier 1 grant MOE2012-T1-001-094.
Authors' adresses: M. Henzinger, University of Vienna, Faculty of Computer Science, Währinger Straße 29, 1090 Wien, Austria; email: monika.henzinger@univie.ac.at; S. Krinninger (Current address), University of Salzburg, Department of Computer Sciences, Jakob-Haringer-Straße 2, 5020 Salzburg, Austria; email: sebastian.krinninger@sbg.ac.at; D. Nanongkai (Current address), KTH Royal Institute of Technology, School of Computer Science and Communication (CSC), Lindstedtsvägen 3, SE-100 44 Stockholm, Sweden; email: danupon@gmail.com.

https://doi.org/10.1145/3146550

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* (Lynch 1996; Peleg 2000). It can be used, for instance, to approximate the network diameter and to provide a communication backbone for broadcast, routing, and control. In this article, we study the problem of maintaining a BFS tree from a root node on dynamic distributed networks. Our main interest is repairing a BFS tree as fast as possible after each topology change.

Model. We model the communication network by the CONGEST model (Peleg 2000), one of the major models of (locality-sensitive) distributed computation. Consider a synchronous network of processors modeled by an undirected unweighted graph $G = (V, E)$, where nodes model the processors and edges model the bounded-bandwidth links between the processors. We let V and E denote the set of nodes and edges of G, respectively, and let s be a specified *root node*. For any node u and v, we denote by $d_G(u, v)$ 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 (such as 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 G_0, where nodes communicate on G_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 G_0 and the states of all nodes, inserts and deletes an arbitrary number of edges in G_0. We denote the resulting network by G_1. This is followed by the first recovery stage where we allow nodes to communicate on G_1. After the nodes have finished communicating, the second attack stage starts, followed by the second recovery stage, and so on. 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 as 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 $\{G_0, G_1, \ldots\}$. If some network G_t is not connected, then we define its diameter as the diameter of the connected component containing the root node. Note that $D \leq n$ according to this definition. 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, then it is *decremental*.

Our model highlights two aspects of dynamic networks: (1) How quickly a network can recover its infrastructure after changes and (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, a few assumptions are inherent in our model. First, it is assumed that the network remains static in each recovery stage. This assumption is often used (e.g., Korman (2008), Hayes et al. (2012), Krizanc et al. (2004), and Malpani et al. (2000)) and helps to emphasize 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 (Peleg 2000; Awerbuch 1985). Furthermore, we consider amortized update time, which is similar in spirit to the amortized communication complexity heavily studied earlier (e.g., Awerbuch et al. (2008)). Finally, the results in this article 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 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 anymore, for example, the network diameter becomes too large. Partially dynamic algorithms for maintaining a BFS tree, which in turn maintains the approximate 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 Peleg (2000, Definition 3.2.2).

Definition 1.1 (Approximate BFS Tree). For any $\alpha \geq 1$, an *α-approximate BFS tree* of an unweighted undirected graph G with respect to a given root s is a spanning tree T of the connected component containing s, such that for every node v connected to s, $d_T(v, s) \leq \alpha d_G(v, s)$. If $\alpha = 1$, then T is an *(exact) BFS tree*.

Note that, for any spanning tree T of G, $d_T(v, s) \geq d_G(v, s)$. Our goal 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 G_t and, for each neighbor u of v, v knows if u is its parent or child in T_t. Note that for convenience, we will usually consider $d_G(v, s)$, the distance of v *to* the root, instead of $d_G(s, v)$, the distance of v *from* the root. In an undirected graph both values are the same.

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., Peleg (2000) and Lynch (1996)), we can do this in time $O(D_t)$, where D_t is the diameter of the graph G_t. Thus, the update time is $O(D)$.

Results. Our main results are partially dynamic algorithms that break the naive update time of $O(D)$ in the long term. 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 $\omega(n/D)$ edge changes. To be precise, the amortized update time over q edge changes is

$$O\left(\frac{n^{1/3}D^{2/3}}{\epsilon^{2/3}q^{1/3}}\right) \text{ and } O\left(\frac{n^{1/5}D^{4/5}}{\epsilon q^{1/5}}\right)$$

in the incremental and decremental setting, respectively. For the particular case of $q = \Omega(n)$, we get amortized update times of $O(D^{2/3}/\epsilon^{2/3})$ and $O(D^{4/5}/\epsilon)$ for the incremental and decremental cases, respectively. Our algorithms do not require any prior knowledge about the dynamic network, for example, D and q. We have formulated the algorithms for a setting that allows insertions or deletions of edges. The guarantees of our algorithms also hold when we allow insertions or deletions of *nodes*, where the insertion of a node also inserts all its incident edges and the deletion of a node also deletes all its incident edges. In the running time, the parameter q then counts the number of node insertions or node deletions, respectively.

We note that, while there is no previous literature on this problem, one can parallelize the algorithm of Even and Shiloach (1981) (see also King (1999) and Roditty and Zwick (2011)) 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. They are faster than the Even-Shiloach algorithm when $q = \omega(\epsilon n\sqrt{D})$ (incremental) and $q = \omega(\epsilon^{7/12} n D^{1/6})$ (decremental).

In the sequential (usual RAM) model, our technique also gives an $(1 + \epsilon)$-approximation algorithm for the incremental single-source shortest paths (SSSP) problem with an amortized update time of $O(mn^{1/4} \log n/\sqrt{\epsilon q})$ 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 (1981) from the 1980s, with $O(mn/q)$ amortized update time, was known. No further progress has been made in the last three decades. Roditty and Zwick (2011) provided an explanation for this by showing that the algorithm of Even and Shiloach (1981) is likely to be the fastest combinatorial *exact* algorithm, assuming that there is no faster combinatorial algorithm for Boolean matrix multiplication. More recently, Henzinger et al. (2015) showed that by assuming a different conjecture, called Online Matrix-Vector Multiplication Conjecture, this statement can be extended to any algorithm (including non-combinatorial ones). Bernstein and Roditty (2011) showed that, in the decremental setting, this bound can be broken if some approximation is allowed. Our result is the first one of the same spirit in the *incremental* setting for deterministic algorithms; that is, we break the bound of Even and Shiloach for the case $q = o(n^{3/2})$, which in particular applies when $m = o(n^{3/2})$. The techniques introduced in this article (first presented in the preliminary version (Henzinger et al. 2013)), together with techniques from Henzinger et al. (2016), also led to a decremental algorithm (Henzinger et al. 2014a) that improves the result of Bernstein and Roditty (2011). We finally obtained a near-optimal algorithm in the decremental setting (Henzinger et al. 2014b), which is a significant improvement over (Bernstein and Roditty 2011). In terms of deterministic algorithms, Bernstein and Chechik have recently presented improved incremental and decremental algorithms for dense (Bernstein and Chechik 2016) and sparse graphs (Bernstein and Chechik 2017). For very sparse graphs with $m = \Theta(n)$, the incremental algorithm in this article still remains the fastest.

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 1970s; see, for example, Awerbuch et al. (2008) 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 Korman and Peleg (2008) where the main goal is to optimize the amortized update time using static algorithms in the recovery stages. The model in Korman and Peleg (2008) is still slightly different from ours in terms of allowed changes. For example, the model in Korman and Peleg (2008) 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 is the *controlled dynamic model* (e.g., Korman and Kutten (2013) and Afek et al. (1996)), 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 well, since 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, for example, Korman (2008), Hayes et al. (2012), Krizanc et al. (2004), and Malpani et al. (2000) whose main interest is to determine how fast a network can recover from changes using static algorithms in the recovery stages, and those in, for example, Awerbuch et al. (2008), Afek et al. (1987), and Elkin (2007), which focus on the amortized cost per edge change. Variations of partially dynamic distributed networks have also been considered (e.g., Italiano (1991), Ramarao and Venkatesan (1992), and Cicerone et al. (2007, 2010)).

The problem of constructing a BFS tree has been studied intensively in various distributed settings for decades (see Peleg (2000, Chapter 5) and Lynch (1996, Chapter 4) and references therein). The studies were also extended to more sophisticated structures such as minimum spanning trees (e.g., Garay et al. (1998), Kutten and Peleg (1998), Peleg and Rubinovich (2000), Elkin (2006), Lotker et al. (2006, 2005), Kor et al. (2013), Das Sarma et al. (2012), Elkin et al. (2014), and Steiner trees (Khan et al. 2012). These studies usually focus on *static* networks, that is, they assume that the network never changes and one wants 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 attention (e.g., Awerbuch et al. (2008)) has previously been given to the problem of *maintaining a spanning tree*. In a seminal article by Awerbuch et al. (2008), 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 (Afek et al. 1987).[1] Our result is in the same spirit as Awerbuch et al. (2008) 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., Alstrup et al. (2005); Italiano and Ramaswami (1998); Nardelli et al. (2001, 2003); Salvo and Proietti (2007); Ito et al. (2005); Das et al. (2010); Gfeller (2012)) and distributed (e.g., Gfeller et al. (2011); Flocchini et al. (2006)) 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 Even and Shiloach (1981), which might be optimal (Roditty and Zwick 2011; Henzinger et al. 2015), has recently been improved by randomized decremental approximation algorithms (Bernstein and Roditty 2011; Henzinger et al. 2014a, 2014b), and we achieve a similar result in the incremental setting with a deterministic algorithm. Since our algorithms use the algorithm of Even and Shiloach (1981) as a subroutine, we formally state its guarantees in the following. As mentioned above, this algorithm has not been considered in the distributed model before, but its analysis from the sequential model immediately

[1]A variant of their algorithm was later implemented as a part of the PARIS networking project at IBM (Cidon et al. 1995) and slightly improved (Kutten and Porat 1999).

carries over to the distributed model.[2] Since we will need this result later in this article, we state it here.

Theorem 1.2 ((Even and Shiloach 1981)). *There is a partially dynamic algorithm for maintaining a shortest paths tree from a given root node up to depth $X \leq n$ under edge insertions (deletions) in an unweighted, undirected graph. Its total running time over q insertions (deletions) is $O(mX)$ in the sequential model and $O(n \min(X, D) + q)$ in the distributed model.*

2 MAIN TECHNICAL IDEA

All our algorithms are based on a simple idea of modifying the algorithm of Even and Shiloach (1981) with lazy updates, which we call *lazy Even-Shiloach tree*. 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 κ and δ be parameters. For every recovery stage t, we maintain a tree T_t, such that $d_{G_t}(v, s) \leq d_{T_t}(v, s) \leq d_{G_t}(v, s) + \kappa\delta$ for every node v. We will do this by recomputing a BFS tree from scratch repeatedly, specifically $O(q/\kappa + nD/\delta^2)$ times during q updates.

During the preprocessing, our algorithm constructs a BFS tree of G_0, denoted by T_0. This means that every node u knows its parent and children in T_0 and the value of $d_{T_0}(u, s)$. Suppose that, in the first attack stage, an edge is inserted, say (u, v) where $d_{G_0}(u, s) > d_{G_0}(v, s)$. As a result, the distance from u to s might decrease, that is, $d_{G_1}(u, s) < d_{G_0}(u, s)$. In this case, the distances from s to some other nodes (e.g., the children of u 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 u to s decreases by at least δ; otherwise, we simply do nothing! In the latter case, we say that *u is lazy*. Additionally, we regularly "clean up" by recomputing the BFS tree after each κ 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 δ, since we could have decreased the distance of v and other nodes by at most δ, 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 each κ insertions, the additive error will be at most $\kappa\delta$ as claimed.

To bound 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 could also be caused by some node v, whose distance to s decreases by at least δ. Since every time a node v causes a recomputation, its distance decreases by at least δ, and since $d_{G_0}(v, s) \leq D$, v will cause the recomputation at most D/δ times. This naive argument shows that there are nD/δ 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 any neighbor of v, say v', to s also decreases by at least $\delta - 1$. Similarly, the distance of any neighbor of v' to s decreases by at least $\delta - 2$, and so on. This leads to the conclusion that one recomputation corresponds to $(\delta + (\delta - 1) + (\delta - 2) + \cdots) = \Omega(\delta^2)$ distance decreases. Thus, the number of recomputations is at most nD/δ^2. Combining the two bounds,

[2]In the sequential model, the algorithm has to perform work proportional to the degree of each node whose distance to the root decreases (increases). Assume we are interested in a shortest paths tree up to depth X. As each node's distance to the root can increase (decrease) at most X times, the total running time is $O(mX)$. In the distributed model, sending a message to all neighbors takes one round, and thus we only charge constant time to each level increase (decrease) of a node, resulting in a total time of $O(n \min(X, D) + q)$. The additional q comes from the fact that we have to spend constant time per insertion (deletion), which in the sequential model is dominated by other running time aspects.

we get that the number of BFS tree computations is $O(q/\kappa + nD/\delta^2)$ as claimed above. We get a bound on the total time when we multiply this number by the time needed for a single BFS tree computation. In the sequential model this takes time $O(m)$, where m is the final number of edges, and in the distributed model this takes time $O(D)$, where D is the dynamic diameter of the network.

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 (1981) 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 κ and δ appropriately, the above incremental algorithm immediately gives total update times of $O(mn^{2/5}q^{2/5}/\epsilon^{2/5})$ and $O(q^{2/5}n^{3/5}D^{4/5}/\epsilon^{2/5})$ in the sequential and distributed model, respectively. To obtain the running time bounds claimed in the introduction of this article, we need one more idea called *layering*, where we use different values of δ and κ depending on the distance of each node to s. In the decremental setting, the situation is much more difficult, mainly because it is expensive 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, for example, constructing a virtual tree (in which edges sometimes represent paths).

3 INCREMENTAL ALGORITHM

In this section, we 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 sequential model and the distributed model, respectively. For simplicity, we assume that the initial graph is connected. In Section 3.4, we explain how to remove this assumption.

3.1 General Framework

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 δ, our algorithm keeps its tree T_0 and accepts an *additive error* of δ for every node. When the decrease exceeds δ, our algorithm starts a new phase and recomputes the BFS tree. It also starts a new phase after each κ 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 δ and κ, 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 so far.

As we show below the algorithm gives the desired additive 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 κ edges have been inserted, the additive error is at most $\kappa\delta$.

Lemma 3.1 (Additive Approximation). *For every $\kappa \geq 1$ and $\delta \geq 1$, Algorithm 1 provides the following approximation guarantee for every node x, such that $d_{G_0}(x, s) \leq X$:*

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

ALGORITHM 1: Incremental Algorithm

```
Procedure INSERT $(u, v)$
    $k \leftarrow k + 1$;
    if $k = \kappa$ then INITIALIZE();
    if $d_{G_0}(u, s) > d_{G_0}(v, s) + \delta$ then INITIALIZE();

Procedure INITIALIZE()                                        // Start new phase
    $k \leftarrow 0$;
    Compute BFS tree $T$ of depth $X$ rooted at $s$ and current distances $d_{G_0}(\cdot, s)$;
```

Proof. The algorithm can only provide the approximation guarantee 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 $d_G(x, s) \leq d_{G_0}(x, s)$, because G is the result of inserting edges into G_0. In the following, we argue about the second inequality.

Consider the shortest path $\pi = x_l, x_{l-1}, \ldots x_0$ 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 $x_j, x_{j-1}, \ldots, x_0$ that were inserted since the beginning of the current phase.

Claim 3.2. *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 κ 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 π, 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) is 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 π 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. □

Remark 3.3. In the proof of Lemma 3.1, we need the property that at most κ edges on the shortest path to the root have been inserted since the beginning of the current phase. If we allow inserting $\kappa/2$ nodes (together with their set of incident edges), then we will see at most κ inserted edges on the shortest path to the root as each node appears at most once on this path and contributes at most two incident edges. Thus, we can easily modify our algorithms to deal with node insertions with the same approximation guarantee and asymptotic running time.

If an edge (u, v) is inserted into the graph, such that the inequality $d_{G_0}(u, s) \leq d_{G_0}(v, s) + \delta$ does not hold (and subsequently the algorithm calls the procedure `initialize`), then we cannot guarantee our bound on the additive error anymore. Nevertheless, the algorithm makes progress in some sense: After the insertion, u has an edge 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 δ since the beginning of the current phase. Thus, testing whether $d_{G_0}(u, s) > d_{G_0}(v, s) + \delta$ is a fast way of testing whether $d_{G_0}(u, s) \geq d_G(u, s) + \delta$, that is, whether the distance between u and s has decreased so much that a rebuild is necessary.

Lemma 3.4. *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. We have 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 $d_{G_0}(u, s) \geq d_G(u, s) + \delta$. □

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

Proof. Let π denote the shortest path from y to s of length $d_H(y, s)$ in H. We first bound the distance change for single nodes.

Claim 3.6. *For every node x on π, we have $d_H(x, s) \geq d_{H'}(x, s) + \delta - d_H(x, y) - d_{H'}(x, y)$.*

Proof of Claim. By the triangle inequality, we have $d_{H'}(x, s) \leq d_{H'}(x, y) + d_{H'}(y, s)$, which is equivalent to $d_{H'}(y, s) \geq d_{H'}(x, s) - d_{H'}(x, y)$. By this inequality and the fact that x lies on π, the shortest path from y to s in H, we have

$$d_H(y, x) + d_H(x, s) = d_H(y, s) \geq d_{H'}(y, s) + \delta \geq d_{H'}(x, s) - d_{H'}(x, y) + \delta.$$

Since $d_H(y, x) = d_H(x, y)$ the claimed inequality follows. □

From the claim and the fact that $d_{H'}(x, y) \leq d_H(x, y)$, we conclude that

$$\begin{aligned}\sum_{x \in \pi, d_H(x,y) < \delta/2} d_H(x, s) &\geq \sum_{x \in \pi, d_H(x,y) < \delta/2} (d_{H'}(x, s) + \delta - 2d_H(x, y)) \\ &= \left(\sum_{x \in \pi, d_H(x,y) < \delta/2} d_{H'}(x, s) \right) + \left(\sum_{x \in \pi, d_H(x,y) < \delta/2} (\delta - 2d_H(x, y)) \right)\end{aligned}$$

$$\geq \left(\sum_{x \in \pi, d_H(x,y) < \delta/2} d_{H'}(x,s) \right) + \left(\sum_{j=1}^{\lfloor \delta/2 \rfloor} (\delta - 2j) \right)$$

$$= \left(\sum_{x \in \pi, d_H(x,y) < \delta/2} d_{H'}(x,s) \right) + \delta(\lfloor \delta/2 \rfloor) - \lfloor \delta/2 \rfloor (\lfloor \delta/2 \rfloor + 1)$$

$$= \left(\sum_{x \in \pi, d_H(x,y) < \delta/2} d_{H'}(x,s) \right) + \Omega(\delta^2).$$

Finally, we get

$$\sum_{x \in V} d_H(x,s) = \left(\sum_{x \in \pi, d_H(x,y) < \delta/2} d_H(x,s) \right) + \sum_{x \notin \pi \text{ or } d_H(x,y) \geq \delta/2} \underbrace{d_H(x,s)}_{\geq d_{H'}(x,s)}$$

$$\geq \left(\sum_{x \in \pi, d_H(x,y) < \delta/2} d_{H'}(x,s) \right) + \Omega(\delta^2) + \sum_{x \notin \pi \text{ or } d_H(x,y) \geq \delta/2} d_{H'}(x,s)$$

$$= \left(\sum_{x \in V} d_{H'}(x,s) \right) + \Omega(\delta^2). \qquad \square$$

The quadratic distance decrease 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 3.7 (Running Time). *For every $\kappa \geq 1$ and $\delta \geq 1$, the total update time of Algorithm 1 is $O(T_{\text{BFS}}(X) \cdot (q/\kappa + nX/\delta^2 + 1) + q)$, where $T_{\text{BFS}}(X)$ is an upper bound on the time needed for computing a BFS tree up to depth X.*

Proof. Besides the constant time per insertion, 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 κ, which can happen at most q/κ 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 3.4 and 3.5 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. The overall running time thus is $O(T_{\text{BFS}}(X) \cdot (q/\kappa + nX/\delta^2 + 1) + q)$. The 1-term is just a technical necessity as the BFS tree has to be computed at least once. $\square$

The algorithm above provides an additive approximation. In the following, we turn this into a multiplicative approximation for a fixed distance range. Using a multi-layer approach, we enhance this to a multiplicative approximation for the full distance range in Sections 3.2 (sequential model) and 3.3 (distributed model).

Lemma 3.8 (Multiplicative Approximation). *Let $0 < \epsilon \leq 1$, $X \leq n$, and set $\gamma = \epsilon/4$. If $\gamma^2 qX \geq n$ and $\gamma n X^2 \geq q$, then by setting $\kappa = q^{1/3}X^{1/3}\gamma^{2/3}/n^{1/3}$ and $\delta = n^{1/3}X^{2/3}\gamma^{1/3}/q^{1/3}$, Algorithm 1 has a total update time of*

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

where $T_{\text{BFS}}(X)$ is an upper bound on the time needed for computing a BFS tree up to depth X. Furthermore, it provides the following approximation guarantee: $d_{G_0} \geq d_G(x,s)$ for every node x and $d_{G_0}(x,s) \leq (1+\epsilon) d_G(x,s)$ for every node x, such that $X/2 \leq d_{G_0}(x,s) \leq X$.

Proof. To simplify the notation a bit, we define $A = \kappa\delta$, which gives $A = \gamma X$. By Lemma 3.7, Algorithm 1 runs in time

$$O\left(T_{\text{BFS}}(X) \cdot \left(\frac{q}{\kappa} + \frac{nX}{\delta^2} + 1\right) + q\right).$$

It is easy to check that by our choices of κ and δ the two terms appearing in the running time are balanced, and we get

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

Furthermore, the inequalities $\gamma^2 qX \geq n$ and $\gamma nX^2 \geq q$ ensure that $\kappa \geq 1$ and $\delta \geq 1$.

We now argue that the approximation guarantee holds. By Lemma 3.1, we already know that

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

for every node x, such that $d_{G_0}(x,s) \leq X$. We now show that our choices of κ and δ guarantee 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

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

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 $A \leq \epsilon d_G(x,s)$. □

3.2 Sequential Model

It is straightforward to use the abstract framework of Section 3.1 in the sequential model. First, note that in the sequential model computing a BFS tree takes time $O(m)$, regardless of the depth. We run $O(\log n)$ "parallel" instances of Algorithm 1, where each instance provides a $(1+\epsilon)$-approximation for nodes in some distance range from $X/2$ to X. However, when X is small enough, then instead of maintaining the approximate distance with our own algorithm it is more efficient to maintain the exact distance using the algorithm of Even and Shiloach (1981).

Theorem 3.9. *In the sequential model, there is an incremental* $(1+\epsilon)$*-approximate SSSP algorithm for inserting up to q edges that has a total update time of* $O(mn^{1/4}\sqrt{q}\log n/\sqrt{\epsilon})$ *where m is the number of edges in the final graph. It answers distance and path queries in optimal worst-case time.*

Proof. If $q \leq 8n^{1/2}/\epsilon$, then we recompute a BFS tree from scratch after every insertion. This takes time $O(mq) = O(mq^{1/2}q^{1/2}) = O(mn^{1/4}q^{1/2}/\epsilon^{1/2})$.

If $q > 8n^{1/2}/\epsilon$, then the algorithm is as follows. Let X^* be the smallest power of 2 greater than or equal to $2n^{1/4}q^{1/2}/\epsilon^{1/2}$ (i.e., $X^* = 2^{\lceil \log (2n^{1/4}q^{1/2}/\epsilon^{1/2}) \rceil}$). First, we maintain an Even-Shiloach tree up to depth X^*, which takes time $O(mX^*) = O(mn^{1/4}q^{1/2}/\epsilon^{1/2})$ by Theorem 1.2. Additionally, we run $O(\log n)$ instances of Algorithm 1, one for each $\log X^* \leq i \leq \lceil \log n \rceil$. For the ith instance, we set the parameter X to $X_i = 2^i$ and κ and δ as in Lemma 3.8. Every time we start a new phase for instance i, we also start a new phase for every instance j, such that $j \leq i$. This guarantees that if a node leaves the range $[X_i/2, X_i]$ (which in the incremental model can only happen if the distance to the root goes below $X_i/2$) it will immediately be covered by a lower range. Since the graph is connected, we now have the following property: for every node v with distance more than X^* to s there is at least one index i, such that v is in the range $[X_i/2, X_i]$, that is, at the beginning of the current phase of instance i the distance from v to s was between $X_i/2$ and X_i. By Lemma 3.8, this previous distance is a $(1+\epsilon)$-approximation of the current distance. The algorithm can, at no overhead in asymptotic running time, easily track the smallest i, such that v is in the range $[X_i/2, X_i]$ for every node v.

The cost of starting a new phase for every instance $j \leq i$ is $O(m \log n)$, since we have to construct a BFS tree up to depth X_j for all $j \leq i$. By Lemma 3.8, the running time of the ith instance of Algorithm 1 therefore is $O(mq^{2/3}n^{1/3} \log n/(\epsilon^{2/3}X_i^{1/3}))$, which over all instances gives a running time of

$$O\left(\sum_{\log X^* \leq i \leq \lceil \log n \rceil} \frac{mq^{2/3}n^{1/3} \log n}{\epsilon^{2/3}X_i^{1/3}}\right) = O\left(\frac{mq^{2/3}n^{1/3} \log n}{\epsilon^{2/3}X^{*1/3}}\right) = O\left(\frac{mn^{1/4}q^{1/2} \log n}{\epsilon^{1/2}}\right).$$

Note that for each instance i, Lemma 3.8 only applies if $\gamma^2 qX_i \geq n$ and $\gamma n X_i^2 \geq q$. These two inequalities hold, because q and X^* are large enough:

$$\gamma^2 qX_i = \epsilon^2 qX_i/16 \geq \epsilon^2 qX^*/16 \geq \epsilon^{3/2}q^{3/2}n^{1/4}/8 \geq \epsilon^{3/2}n^{3/4}n^{1/4}/\epsilon^{3/2} = n$$
$$\gamma n X_i^2 = \epsilon n X_i^2/4 \geq \epsilon n (X^*)^2/4 \geq 4\epsilon n^{3/2} q/(4\epsilon) = n^{3/2}q \geq q$$

Finally, we argue that the number q of insertions does not have to be known beforehand. We use a doubling approach for guessing the value of q where the ith guess is $q_i = 2^i$. When the number of insertions exceeds our guess q_i, we simply restart the algorithm and use the guess $q_{i+1} = 2q_i$ from now on. The total running time for this approach is $O(\sum_{i=0}^{\lceil \log q \rceil} mn^{1/4}q_i^{1/2} \log n/\epsilon^{1/2})$, which is $O(mn^{1/4}q^{1/2} \log n/\epsilon^{1/2})$. □

3.3 Distributed Model

In the distributed model, we use the same multi-layer approach as in the sequential model. However, we have to consider some additional details for implementing the algorithm, because not all information is globally available to every node in the distributed model. Computing a BFS tree up to depth X takes time $T_{\text{BFS}} = O(X)$ in the distributed model. In the running time analysis of Lemma 3.7, we thus charge time $O(X)$ to every phase and constant time to every insertion. We now argue that this is enough to implement the algorithm in the distributed model.

After the insertion of an edge (u, v) the nodes u and v have to compare their initial distances $d_{G_0}(u, s)$ and $d_{G_0}(v, s)$. They can exchange these numbers with a constant number of messages, which we account for by charging constant time to every insertion.

The root node s has to coordinate the phases and thus needs to gather some special information. The first cause for starting a new phase is when the level of some node decreases by at least δ. If a node detects a level decrease by at least δ, then it has to inform the root s about the decrease so 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 sending this message is $O(X)$, which we charge to the current phase. Note that, similar to recomputing the BFS tree, this happens in a recovery stage during which no new edges are inserted.

The second cause for starting a new phase is that the number of edge insertions, since the beginning of the current phase, exceeds κ. Therefore, it is necessary that the root s knows the number of edges that have been inserted. We count the number of insertions at the root as follows. After each insertion of an edge (u, v) the node v sends a message to the root to inform it about the edge insertion. We will make sure that this message arrives at the root with small enough delay; in particular each insertion message will arrive at the root after $\kappa/2$ recovery stages. Again, the tree maintained by our algorithm, which has depth at most X, can be used to send the insertion messages to the root. During each recovery stage, we move up all the insertion messages that have not yet arrived at the root along $2X/\kappa$ nodes in the tree (i.e., we decrease the level of each such message by at least $2X/\kappa$). To avoid congestion, we aggregate insertion messages meeting at the same node by simply counting the *number* of insertions. Thus, we need to spend an additional $O(X/\kappa)$ rounds in each recovery stage. In this way, the first insertion message arrives at the root after $\kappa/2$ recovery stages and after κ recovery stages the first $\kappa/2$ messages have arrived Accumulated over κ recovery stages after insertions, the total time for sending the insertion messages is $O(\kappa X/\kappa) = O(X)$, which we charge to the current phase. Thus, to get the same approximation guarantee and the same asymptotic running time as in Section 3.3, we slightly modify the algorithm to start a new phase as soon as the root has been notified of $\kappa/2$ insertions.

Theorem 3.10. *In the distributed model, there is an incremental algorithm for maintaining a $(1 + \epsilon)$-approximate BFS tree under up to q insertions that has a total update time of $O(q^{2/3}n^{1/3}D^{2/3}/\epsilon^{2/3})$, where D is the dynamic diameter.*

Proof. Our algorithm consists of $O(\log D)$ layers. For each $0 \leq i \leq \lceil \log D \rceil$, we set $X_i = 2^i$ and do the following: (1) If $q \leq 16n/(\epsilon^2 X_i)$, then we recompute a BFS tree up to depth X_i from scratch after every insertion. (2) If $q > 16n/(\epsilon^2 X_i)$ and $X_i \leq 4\sqrt{q}/\sqrt{\epsilon n}$, then we maintain an Even-Shiloach tree up to depth X_i. (3) If $q > 16n/(\epsilon^2 X_i)$ and $X_i > 4\sqrt{q}/\sqrt{\epsilon n}$, then we run an instance of Algorithm 1 with parameters $X_i = 2^i$ and κ_i and δ_i as in Lemma 3.8. We use the following slight modification of Algorithm 1: every time a new phase starts for instance i, we re-initialize all instances j of Algorithm 1, such that $j \leq i$ by computing a BFS tree up to depth X_j. Note that if D is not known in advance, our algorithm can simply increase the number of layers until the BFS tree computed at the initialization of the current layer contains all nodes of the graph.

We first argue that this algorithm provides a $(1 + \epsilon)$-approximation. The algorithm maintains the exact distances for all nodes that are in distance at most $16n/(\epsilon^2 q)$ or $4\sqrt{q}/\sqrt{\epsilon n}$ from the root as in these cases the distances are obtained by recomputing the BFS tree from scratch or by the Even-Shiloach tree. For all other nodes, we have to argue that our multi-layer version of Algorithm 1 provides a $(1 + \epsilon)$-approximation. Note that for each instance i the result of Lemma 3.8 only applies if $\gamma^2 q X_i \geq n$ and $\gamma n X_i^2 \geq q$. These two inequalities hold, because q and X_i are large enough:

$$\gamma^2 q X_i = \epsilon^2 q X_i/16 \geq \epsilon^2(16n/(\epsilon^2 X_i))X_i/16 = n$$
$$\gamma n X_i^2 = \epsilon n X_i^2/4 \geq \epsilon n(4\sqrt{q}/\sqrt{\epsilon n})^2/4 = q.$$

In each instance i, the approximation guarantee of Lemma 3.8 holds for all nodes whose distance to the root was between $X_i/2$ and X_i since the last initialization of instance i. Every time we re-initialize instance i, some nodes that before were in the range $[X_i/2, X_i]$ might now have a smaller distance and will thus not be "covered" by instance i anymore. By re-initializing all instances $j \leq i$ as well, we guarantee that such nodes will immediately be "covered" by some other instance of the algorithm (or by the exact BFS tree we maintain for small depths). Since the graph is connected, we thus have the following property: for every node v with distance more than $4\sqrt{q}/\sqrt{\epsilon n}$ to the root there is an index i, such that at the beginning of the current phase of instance i the distance from v to the root was between $X_i/2$ and X_i. By Lemma 3.8 this previous distance is a $(1+\epsilon)$-approximation of the current distance.

We will now bound the running time. We will argue that the running time in every layer i is $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3})$. If the number of insertions is at most $q \leq 16n/(\epsilon^2 X_i)$, then computing a BFS tree from scratch up to depth X_i after very insertion takes time $O(qX_i)$ in total, which we can bound as follows:

$$qX_i = q^{2/3}q^{1/3}X_i = \frac{q^{2/3}n^{1/3}X_i^{2/3}}{\epsilon^{2/3}}.$$

By Theorem 1.2, maintaining an Even-Shiloach tree up to depth $X_i \leq 4\sqrt{q}/\sqrt{\epsilon n}$ takes time $O(nX_i) = O(\sqrt{qn}/\sqrt{\epsilon})$. Since we only do this in the case $q > 16n/(\epsilon^2 X_i)$, we can use the inequality

$$n < \frac{\epsilon^2 q X_i}{16} \leq \frac{qX_i^4}{\epsilon}$$

to obtain

$$nX_i = \frac{\sqrt{qn}}{\sqrt{\epsilon}} = \frac{n^{1/3}n^{1/6}\sqrt{q}}{\sqrt{\epsilon}} \leq \frac{n^{1/3}q^{1/6}X_i^{4/6}\sqrt{q}}{\sqrt{\epsilon}\cdot\epsilon^{1/6}} = \frac{q^{2/3}n^{1/3}X_i^{2/3}}{\epsilon^{2/3}}.$$

Finally, we bound the running time of our slight modification of Algorithm 1 in layer i. Every time we start a new phase in layer i, we re-initialize the instances of Algorithm 1 in all layers $j \leq i$. The re-initialization in each layer j takes time $O(X_j)$ as we have to compute a BFS tree up to depth X_j. Thus, the cost of starting a new phase in layer i is proportional to

$$\sum_{0\leq j\leq i} X_j = \sum_{0\leq j\leq i} 2^j \leq 2^{i+1} = 2X_i,$$

which asymptotically is the same as only the time needed for computing a BFS tree up to depth X_i. Thus, by Lemma 3.8 the running time of instance i of Algorithm 1 is $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3} + q)$. Since $q \leq \epsilon n X_i^2/4$ as argued above, we have $q \leq nX_i^2$, and thus

$$\frac{q^{2/3}n^{1/3}X_i^{2/3}}{\epsilon^{2/3}} \geq q^{2/3}n^{1/3}X_i^{2/3} = q^{2/3}(nX_i^2)^{1/3} \geq q^{2/3}q^{1/3} = q.$$

It follows that the running time of instance i is $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3})$ and the total running time over all layers is

$$O\left(\sum_{0\leq i\leq\lceil\log D\rceil} \frac{q^{2/3}n^{1/3}X_i^{2/3}}{\epsilon^{2/3}}\right) = O\left(\sum_{0\leq i\leq\lceil\log D\rceil} \frac{q^{2/3}n^{1/3}(2^i)^{2/3}}{\epsilon^{2/3}}\right) = O\left(\frac{q^{2/3}n^{1/3}D^{2/3}}{\epsilon^{2/3}}\right).$$

By using a doubling approach for guessing the value of q, we can run the algorithm with the same asymptotic running time without knowing the number of insertions beforehand. □

3.4 Removing the Connectedness Assumption

The algorithm above assumes that the graph is connected. We now explain how to adapt the algorithm to handle graphs where this is not the case.

Note that an insertion might connect one or several nodes to the root node. For each newly connected node, every path to the root goes through an edge that has just been inserted. In such a situation, we extend the tree maintained by the Algorithm 1 by performing a breadth-first search among the newly connected nodes. Using this modified tree, the argument of Lemma 3.1 to prove the additive approximation guarantee still goes through. Note that each node can become connected to the root node at most once. Thus, we can amortize the cost of the breadth-first searches performed to extend the tree over all insertions.

This results in the following modification of the running time of Lemma 3.7: In the sequential model, we have an additional cost of $O(m)$ as each edge has to be explored at most once in one of the breadth-first searches. In the distributed model, we have an additional cost of $O(n)$ as every node is explored at most once in one of the breadth-first searches. The total update time of the $(1+\epsilon)$-approximation in the sequential model (Theorem 3.9) clearly stays unaffected from this modification, as we anyway have to consider the cost of $O(m)$ for computing a BFS tree. In the distributed model the argument is as follows. In the proof of Theorem 3.10, we bound the running time of each instance i of Algorithm 1 by $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3})$. Since q and X_i satisfy the inequality $q > 16n/(\epsilon^2 X_i) \geq n/X_i$, we have $q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3} \geq q^{2/3}n^{1/3}X_i^{2/3} \geq n$. Thus, the additional $O(n)$ is already dominated by $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3})$ and the total update time stays the same as stated in Theorem 3.10. Note that if the number of nodes n is not known in advance because of the graph not being connected, we can use a doubling approach to guess the right range of n.

Insert ê

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 how 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 a node. As in the incremental setting, the algorithm has the parameters κ, δ, and X.

The Procedure `repairTree` of Algorithm 2 either computes a (weighted) tree T that approximates the true distances with an additive error of $\kappa\delta$, or it reports a distance increase by at least δ since the beginning of the current phase. Let T_0 denote the BFS tree computed at the beginning of the current phase, let F' be the forest resulting from removing those edges from T_0 that have already been deleted in the current phase, and the let U be the set of nodes (except for s) that have no parent in F'. After every deletion, the Procedure `repairTree` tries to construct a tree T by starting with the forest F'. Every node $u \in U$ tries to find a "good" node v to reconnect to and if successful will use v as its new parent with a weighted edge (u, v) (whose weight corresponds to the current distance between u and v). Algorithm 2 imposes two conditions (Line 19) on a "good" node v. Condition (1) avoids that the reconnection introduces any cycles and Condition (2) guarantees that the error introduced by each reconnection is at most δ and that a suitable node v can be found in distance at most δ to u. As δ is relatively small, this is the key to efficiently finding such a node. In the following, we denote the distance between two nodes x and y in a graph F with weighted edges by $d_F^{\mathrm{w}}(x, y)$. Note that here we have formulated the algorithm in a way such that the Procedure `repairTree` always starts with a forest F' that is the result of removing all edges from T_0 that have been deleted so far in the current phase, regardless of trees previously computed by the Procedure `repairTree`.

4.1 Analysis of Procedure for Repairing the Tree

In the following, we first analyze only the Procedure `repairTree`. Its guarantees can be summarized as follows.

LEMMA 4.1. *The Procedure* `repairTree` *of Algorithm 2 either reports "distance increase by at least δ" and guarantees that there is a node x with $d_{G_0}(x, s) \leq X$, such that*

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

or it returns a tree T such that for every node x with $d_{G_0}(x, s) \leq X$, we have

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

ALGORITHM 2: Decremental Algorithm

```
// At any time, $T_0$ is the BFS tree computed at the beginning of the current phase,
   $F'$ is the forest resulting from removing all deleted edges from $T_0$ and $T$ is the
   current approximate BFS tree
Procedure DELETE $(u, v)$
    $k \leftarrow k + 1$;
    if $k = \kappa$ then
        INITIALIZE();
    else
        Remove edge $(u, v)$ from $F'$;
        REPAIRTREE();
        if REPAIRTREE() reports distance increase by at least $\delta$ then INITIALIZE();

Procedure Initialize()                                              // Start new phase
    $k \leftarrow 0$;
    Compute BFS tree $T_0$ of depth $X$ rooted at $s$;
    Compute current distances $d_{G_0}(\cdot, s)$;
    $T \leftarrow T_0$;
    $F' \leftarrow T_0$;

Procedure RepairTree()
    $F \leftarrow F'$;
    $U \leftarrow \{u \in V \mid u$ has no parent in $F$ and $u \neq s\}$;
    foreach $u \in U$ do                                            // Search process
        Perform breadth-first search from $u$ up to depth $\delta$ and try to find a node $v$, such that
        (1) $d_{G_0}(v, s) < d_{G_0}(u, s)$ and (2) $d_G(u, v) \leq \delta$;
        if such a node $v$ could be found then
            Add edge $(u, v)$ of weight $d_F^{\mathrm{w}}(u, v) = d_G(u, v)$ to $F$;
        else
            return "distance increase by at least $\delta$";
    $T \leftarrow F$;
```

It runs in time $O(\kappa\delta)$ after every deletion.

We first observe that if the Procedure `repairTree` returns a graph, this graph is actually a tree. The input of the procedure is the forest F' obtained from removing some edges from the BFS tree T_0. In this forest, we have $d_{G_0}(v, s) = d_{G_0}(u, s) - 1$ for every child u and parent v. In the Procedure `repairTree`, we add, for every node u that is missing a parent, an edge to a parent v, such that $d_{G_0}(v, s) < d_{G_0}(u, s)$. Thus, the decreasing label $d_{G_0}(v, s)$ for every node v guarantees that T is a tree.

LEMMA 4.2. *The graph T computed by the Procedure* `repairTree` *is a tree.*

We will show next that the Procedure `repairTree` is either successful, that is, every node in U finds a new parent, or the algorithm makes progress, because there is some node whose distance to the root has increased significantly.

LEMMA 4.3. *For every node $u \in U$, if $d_G(u, s) < d_{G_0}(u, s) + \delta$, then there is a node $v \in V$, such that*

(1) $d_{G_0}(v,s) < d_{G_0}(u,s)$ *and*
(2) $d_G(u,v) \leq \delta$.

Proof. If $d_G(u,s) \leq \delta$, then set $v = s$. As $d_{G_0}(s,s) = 0$ and $u \neq s$, this satisfies both conditions.

If $d_G(u,s) > \delta$, then consider the shortest path from u to s in G and define v as the node that is in distance δ from u on this path, that is, such that $d_G(v,s) = d_G(u,s) - \delta$. We then have

$$d_{G_0}(v,s) \leq d_G(v,s) = d_G(u,s) - \delta < d_{G_0}(u,s) + \delta - \delta = d_{G_0}(u,s). \qquad \square$$

Note that in the proof above, we know exactly which node v we can pick for every node $u \in U$. In the algorithm however the node u does not know its shortest path to s in the current graph and thus it would be expensive to specifically search for the node v on the shortest path from u to s defined above. However, we know that v is contained in the local search performed by u. Therefore, u either finds v or some other node that fulfills Conditions (1) and (2).

We now show that every reconnection made by the Procedure `repairTree` adds an additive error of δ, which sums up to $\kappa\delta$ for at most κ reconnections (one per previous edge deletion).

Lemma 4.4. *For the tree T computed by the Procedure* `repairTree` *and every node x, such that $d_{G_0}(x,s) \leq X$, we have*

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

Proof. We call the weighted edges inserted by the Procedure `repairTree` *artificial edges*. In the tree T there are two types of edges: those that were already present in the BFS tree T_0 from the beginning of the current phase and artificial edges added in the Procedure `repairTree`.

First, we prove the inequality $d_G(x,s) \leq d_T^{\mathrm{w}}(x,s)$. Consider the unique path from x to s in the tree T consisting of the nodes $x = x_l, x_{l-1}, \dots x_0 = s$. 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 $d_T^{\mathrm{w}}(x_{j+1}, x_j) = 1 = d_G(x_{j+1}, x_j)$, or was inserted later by the algorithm, which means that $d_T^{\mathrm{w}}(x_{j+1}, x_j) = d_G(x_{j+1}, x_j)$. This means that in any case we have $d_T^{\mathrm{w}}(x_{j+1}, x_j) = d_G(x_{j+1}, x_j)$, and therefore we get

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

Second, we prove the inequality $d_T^{\mathrm{w}}(x,s) \leq d_{G_0}(x,s) + \kappa\delta$. Consider the shortest path $\pi = x_l, x_{l-1}, \dots, x_0$ from x to s in T, where $x_l = x$ and $x_0 = s$. Let S_j (with $0 \leq j \leq l$) denote the number of artificial edges on the subpath $x_j, x_{j-1}, \dots x_0$. For each edge deleted so far, π contains at most one artificial edge. Therefore, we have $S_j \leq \kappa$ for all $0 \leq j \leq l$. Now consider the following claim.

Claim 4.5. *For every $0 \leq j \leq l$, we have $d_T^{\mathrm{w}}(x_j, s) \leq d_{G_0}(x_j, s) + S_j\delta$.*

Assuming the truth of the claim, the desired inequality follows straightforwardly, since $x_l = x$, $x_0 = s$, and $S_l \leq \kappa$.

In the following, we prove the claim by induction on j. In the induction base, we have $j = 0$ and thus $x_j = s$ and $S_j = 0$. The inequality then trivially holds due to $d_T^{\mathrm{w}}(s,s) = 0$. 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 $d_T^{\mathrm{w}}(x_{j+1}, x_j) = d_{G_0}(x_{j+1}, x_j) = 1$. 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

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

The second case is that (x_{j+1}, x_j) is an artificial edge. In that case, we have $d_T^{\mathrm{w}}(x_{j+1}, x_j) = d_G(x_{j+1}, x_j)$ 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:

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

□

Remark 4.6. In the proof of Lemma 4.4, we need the property that after up to κ edge deletions there are at most κ "artificial" edges on the shortest path to the root in T. This also holds if we allow deleting nodes (together with their set of incident edges). Thus, we can easily modify our algorithm to deal with node deletions with the same approximation guarantee and asymptotic running time.

To finish the proof of Lemma 4.1, we analyze the running time of the Procedure `repairTree` and clarify some implementation details for the distributed setting. In the search process, every node $u \in U$ tries 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 δ using breadth-first search, which takes time $O(\delta)$ for a single node. Whenever local searches of nodes in U "overlap" and two messages have to be sent over an edge, we arbitrarily allow to send one of these messages and delay the other one to the next round. As there are at most κ nodes in U, we can simply bound the time needed for all searches by $O(\kappa\delta)$.

Weighted Edges. The tree computed by the algorithm contains weighted edges. Such an edge e corresponds to a path π 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 π if a message is sent over v as part of the weighted edge e.

Broadcasting Deletions. The nodes that do not have a parent in F' before the procedure `repairTree` starts do not necessarily know that a new edge deletion has happened. Such a node only has to become active and do the search if there is a change in its neighborhood within distance δ, otherwise it can still use the weighted edge in the tree T that it previously used, because the two 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 δ about this event. This takes time $O(\delta)$ per deletion, which is within our projected running time.

4.2 Analysis of Decremental Distributed Algorithm

The Procedure `repairTree` provides an additive approximation of the shortest paths and a means for detecting that the distance of some node to s has increased by at least δ since the beginning of the current phase. Using this procedure as a subroutine, we can provide a running time analysis for the decremental algorithm that is very similar to the one of the incremental algorithm.

Lemma 4.7. *For every* $X \geq 1$, $\kappa \geq 1$, *and* $\delta \geq 1$ *the total update time of Algorithm 2 is* $O(qX/\kappa + nX^2/\delta^2 + q\kappa\delta + n)$ *and it provides the following approximation guarantee: If* $d_{G_0}(x, s) \leq X$, *then*

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

Proof. Using the distance increase argument of Lemma 3.5, we can bound the number of phases by $O(q/\kappa + nX^2/\delta^2)$. To every phase, we charge a running time of $O(X)$, which is the time needed for computing a BFS tree up to depth X at the beginning of the phase. Additionally, we charge a running time of $\kappa\delta$ to every deletion since the Procedure `repairTree`, which is called after every deletion, has a running time of $O(\kappa\delta)$ by Lemma 4.1.

As in the incremental distributed algorithm, we have to enrich the decremental algorithm with a mechanism that allows the root node to coordinate the phases. We explain these implementation details and analyze their effects on the running time in the following.

Reporting Distance Increase. When a node v detects a distance increase by more than δ, it tries to inform the root about the distance increase by sending a special message. It sends the message to all nodes in distance at most $2X$ from v in a breadth-first manner, which takes time $O(X)$. If the root is among these nodes, then the root initiates a new phase and the cost of $O(X)$ is charged to the new phase. Otherwise, the nodes in distance at most X from v know that their distance to the root is more than X. In that case in particular all nodes in the subtree of v in F' have received the message and know that their distance to the root is more than X now. All nodes that are in distance at most X from v do not have to participate in the algorithm anymore. Thus, we can charge the time of $O(X)$ for sending the message to these at least X nodes. This gives a one-time charge of $O(1)$ to every node and adds $O(n)$ to the total update time. A special case is when v becomes disconnected from the root and its new component has size less than X. In that case the time for sending the message to all nodes in the component takes time proportional to the size of the component, which again results in a charge of $O(1)$ to each node.

Counting Deletions. We count the number of deletions at the root as follows. First, observe that we do not have to count those deletions that result in a distance increase by more than δ, because after such an event either a new phase starts or the deletion only affects nodes whose distance to the root has increased to more than X after the deletion. The remaining deletions can be counted by sending one message per deletion to the root using the tree maintained by the algorithm, such that each deletion message will arrive at the root after $\kappa/2$ recovery stages. During each recovery stage, we move up all the deletion messages that have not yet arrived at the root along $2X/\kappa$ nodes in the tree. To avoid congestion, we aggregate deletion messages meeting at the same node by simply counting the *number* of deletions. Note that the level of a node in the tree might increase by at most $\kappa\delta$ with every deletion. Therefore, we need to spend time $O(X/\kappa + \kappa\delta)$ during each recovery stage to ensure that every deletion message that has not yet arrived at the root decreases its level in the tree by at least $2X/\kappa$. In this way, the first deletion message arrives after $\kappa/2$ recovery stages and after κ recovery stages the first $\kappa/2$ messages have arrived at the root. This process takes total time $O(X + \kappa^2\delta)$ for κ recovery stages after deletions. We can charge time $O(X)$ to the current phase and time $O(\kappa\delta)$ to each deletion occurring in the phase. Thus, to obtain an additive approximation of exactly $\kappa\delta$, we slightly modify the algorithm to start a new phase as soon as the root has been notified of $\kappa/2$ deletions. □

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 an appropriate choice of the parameters κ and δ for each instance, we can guarantee a $(1+\epsilon)$-approximation.

Lemma 4.8. *Let $0 < \epsilon \leq 1$ and assume that $\epsilon^5 qX/32 \geq n$ and $nX^{3/2} \geq q$. Then, by setting $\kappa = q^{1/5}X^{1/5}/n^{1/5}$ and $\delta = n^{2/5}X^{3/5}/q^{2/5}$, Algorithm 2 runs in time $O(q^{4/5}n^{1/5}X^{4/5})$. Furthermore, it provides the following approximation guarantee: For every node x, such that $d_{G_0}(x,s) \leq X$, we have*

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

and for every node x, such that $d_G(x,s) \geq X/2$, we additionally have

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

Proof. Since $\epsilon^5 qX/32 \geq n$ implies $qX \geq n$, we have $\kappa \geq 1$, and since $nX^{3/2} \geq q$ we have $\delta \geq 1$. It is easy to check that by our choices of κ and δ the three terms in the running time of Lemma 4.7 are balanced, and we get

$$\frac{q}{\kappa} \cdot X = \frac{nX}{\delta^2} \cdot X = q\kappa\delta = q^{4/5}n^{1/5}X^{4/5}.$$

Furthermore, since $qX \geq n$, we have $q^{4/5}n^{1/5}X^{4/5} \geq n^{1/5}(qX)^{4/5} \geq n^{1/5}n^{4/5} = n$, and therefore the running time of the algorithm is $O(q^{4/5}n^{1/5}X^{4/5})$.

We now argue that the approximation guarantee holds. By Lemma 4.7, we already know that

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

for every node x, such that $d_{G_0}(x,s) \leq X$. We now show that our choices of κ and δ guarantee 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 our assumptions, we have $n \leq \epsilon^5 qX/32$, and therefore we get

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

The value $(1+\epsilon)d_{G_0}(x,s)$ is thus a $(1+\epsilon)$-approximation of $d_G(x,s)$. □

Theorem 4.9. *In the distributed model, there is a decremental algorithm for maintaining a $(1+\epsilon)$-approximate BFS tree over q deletions with a total update time of $O(q^{4/5}n^{1/5}D^{4/5}/\epsilon)$, where D is the dynamic diameter.*

Proof. Our algorithm consists of $O(\log D)$ layers. For each $0 \leq i \leq \lceil \log D \rceil$, we set $X_i = 2^i$ and do the following: If $q \leq 32n/(\epsilon^5 X_i)$, then we recompute a BFS tree up to depth X_i from scratch after every deletion. If $q > 32n/(\epsilon^5 X_i)$ and $X_i \leq (q/n)^{2/3}$, then we maintain an Even-Shiloach tree up to depth X_i. If $q > 32n/(\epsilon^5 X_i)$ and $X_i > (q/n)^{2/3}$, then we run an instance of Algorithm 2 with parameters $X_i = 2^i$ and κ_i and δ_i as in Lemma 4.8. Note that D might increase over the course of the algorithm due to edge deletions (or might not be known in advance). Therefore, whenever we initialize the algorithm in the layer with the current largest index, we do a full BFS tree computation. If the depth of the BFS tree exceeds X_i, then we increase the number of layers accordingly and charge the running time of the BFS tree computation to the layer with new largest index.

We first argue that this algorithm provides a $(1+\epsilon)$-approximation. The algorithm maintains the exact distances for all nodes that are in distance at most $32n/(\epsilon^5 q)$ or $(q/n)^{2/3}$ from the root as in these cases the distances are obtained by recomputing the BFS tree from scratch or by the

Even-Shiloach tree. For all other nodes, we have to argue that our multi-layer version of Algorithm 2 provides a $(1+\epsilon)$-approximation. Note that the approximation guarantee of Lemma 3.8 only applies if $\epsilon^5 qX_i/32 \geq n$ and $nX_i^{3/2} \geq q$. These two inequalities hold, because q and X_i are large enough:

$$\epsilon^5 qX_i/32 \geq \epsilon^5(32n/(\epsilon^5 X_i))X_i/32 = n,$$
$$nX_i^{3/2} \geq n((q/n)^{2/3})^{3/2} = q.$$

In each instance i of Algorithm 2, the approximation guarantee of Lemma 3.8 holds for all nodes whose distance to the root at the beginning of the current phase of instance i was at most X_i and whose current distance to the root is at least $X_i/2$. Whenever an instance i starts a new phase, there might be some nodes who before were contained in the tree of instance i, but are not contained in the new tree anymore, because their distance to the root has increased to more than X_i. Since $X_i = X_{i+1}/2$, we know that those nodes will immediately be "covered" by an instance with larger index. Thus, after each recovery stage every node that is connected to the root will be contained in the tree of some instance i, such that the preconditions of Lemma 3.8 apply and thus the distance to the root in that tree provides a $(1+\epsilon)$-approximation. In particular, each node simply has to pick the tree of the smallest index containing it.

We will now bound the running time. We will argue that the running time in every layer i is $O(q^{4/5}n^{1/5}X_i^{4/5}/\epsilon)$. If the number of insertions is at most $q \leq 32n/(\epsilon^5 X_i)$, then computing a BFS tree from scratch up to depth X_i after very insertion takes time $O(qX_i)$ in total, which we can bound as follows:

$$qX_i = q^{4/5}q^{1/5}X_i \leq \frac{q^{4/5}32^{1/5}n^{1/5}X_i^{4/5}}{\epsilon} = O\left(\frac{q^{4/5}n^{1/5}X_i^{4/5}}{\epsilon}\right).$$

By Theorem 1.2 maintaining an Even-Shiloach tree up to depth $X_i \leq (q/n)^{2/3}$ takes time $O(nX_i) = O(q^{2/3}n^{1/3})$. Since we only do this in the case $q > 32n/(\epsilon^5 X_i)$, we can use the inequality

$$n < \frac{\epsilon^5 qX_i}{32} \leq \epsilon^5 qX_i \leq \frac{qX_i^6}{\epsilon^{15/2}}$$

to obtain

$$nX_i \leq q^{2/3}n^{1/3} = q^{2/3}n^{1/5}n^{2/15} \leq q^{2/3}n^{1/5}\frac{q^{2/15}X_i^{4/5}}{\epsilon} = \frac{q^{4/5}n^{1/5}X_i^{4/5}}{\epsilon}.$$

Finally, we use Lemma 4.8 to bound the running time of Algorithm 2 in layer i by $O(q^{4/5}n^{1/5}X_i^{4/5}/\epsilon)$ as well. Thus, the running time over all layers is

$$O\left(\sum_{0\leq i\leq\lceil\log D\rceil}\frac{q^{4/5}n^{1/5}X_i^{4/5}}{\epsilon}\right) = O\left(\sum_{0\leq i\leq\lceil\log D\rceil}\frac{q^{4/5}n^{1/5}(2^i)^{4/5}}{\epsilon}\right) = O\left(\frac{q^{4/5}n^{1/5}D^{4/5}}{\epsilon}\right).$$

By using a doubling approach for guessing the value of q, we can run the algorithm with the same asymptotic running time without knowing the number of deletions beforehand. □

5 CONCLUSION AND OPEN PROBLEMS

In this article, we showed that an approximate breadth-first search spanning tree can be maintained in amortized time per update that is sublinear in the diameter D in partially dynamic distributed networks when amortized over a sufficient number of updates. 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 article, it is also interesting to obtain algorithms with small bounds on message complexity and memory.

We believe that the most interesting open problem is whether the sequential algorithm in this article can be improved to obtain a deterministic incremental algorithm with near-linear total update time. As noted earlier, techniques from this article have led to a *randomized* decremental algorithm with near-linear total update time (Henzinger et al. 2014b) (the same algorithm also works in the incremental setting). Whether this algorithm can be derandomized was left as a major open problem in Henzinger et al. (2014b). As the incremental case is usually easier than the decremental case, it is worth obtaining this result in the incremental setting first.

ACKNOWLEDGMENTS

We thank the reviewers of ICALP 2013 for pointing to related articles and to an error in an example given in a previous version of this article. We also thank one of the reviewers of Transactions on Algorithms for very detailed comments.

REFERENCES

Yehuda Afek, Baruch Awerbuch, and Eli Gafni. 1987. Applying static network protocols to dynamic networks. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS'87)*. 358–370.

Yehuda Afek, Baruch Awerbuch, Serge A. Plotkin, and Michael E. Saks. 1996. Local management of a global resource in a communication network. *J. ACM* 43, 1 (1996), 1–19.

Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2005. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algor.* 1, 2 (2005), 243–264.

Baruch Awerbuch. 1985. Complexity of network synchronization. *J. ACM* 32, 4 (1985), 804–823.

Baruch Awerbuch, Israel Cidon, and Shay Kutten. 2008. Optimal maintenance of a spanning tree. *J. ACM* 55, 4 (2008), 18:1–18:45.

Aaron Bernstein and Shiri Chechik. 2016. Deterministic decremental single source shortest paths: Beyond the $O(mn)$ bound. In *Proceedings of the Symposium on Theory of Computing (STOC'16)*. 389–397.

Aaron Bernstein and Shiri Chechik. 2017. Deterministic partially dynamic single source shortest paths for sparse graphs. In *Proceedings of the Symposium on Discrete Algorithms (SODA'17)*. 453–469.

Aaron Bernstein and Liam Roditty. 2011. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the Symposium on Discrete Algorithms (SODA'11)*. 1355–1365.

Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, and Daniele Frigioni. 2010. Partially dynamic efficient algorithms for distributed shortest paths. *Theor. Comput. Sci.* 411, 7–9 (2010), 1013–1037.

Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, Daniele Frigioni, and Alberto Petricola. 2007. Partially dynamic algorithms for distributed shortest paths and their experimental evaluation. *J. Comput.* 2, 9 (2007), 16–26.

Israel Cidon, Inder S. Gopal, Marc A. Kaplan, and Shay Kutten. 1995. A distributed control architecture of high-speed networks. *IEEE Trans. Commun.* 43, 5 (1995), 1950–1960.

Shantanu Das, Beat Gfeller, and Peter Widmayer. 2010. Computing all best swaps for minimum-stretch tree spanners. *J. Graph Algor. Appl.* 14, 2 (2010), 287–306.

Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. 2012. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.* 41, 5 (2012), 1235–1265.

Michael Elkin. 2006. A faster distributed protocol for constructing a minimum spanning tree. *J. Comput. System Sci.* 72, 8 (2006), 1282–1308.

Michael Elkin. 2007. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'07)*. 185–194.

Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. 2014. Can quantum communication speed up distributed computation? In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'14)*. 166–175.

Shimon Even and Yossi Shiloach. 1981. An on-line edge-deletion problem. *J. ACM* 28, 1 (1981), 1–4.

Paola Flocchini, Antonio Mesa Enriques, Linda Pagli, Giuseppe Prencipe, and Nicola Santoro. 2006. Point-of-failure shortest-path rerouting: Computing the optimal swap edges distributively. *IEICE Trans. Info. Syst.* 89-D, 2 (2006), 700–708.

Juan A. Garay, Shay Kutten, and David Peleg. 1998. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.* 27 (1998), 302–316.

Beat Gfeller. 2012. Faster swap edge computation in minimum diameter spanning trees. *Algorithmica* 62, 1–2 (2012), 169–191.

Beat Gfeller, Nicola Santoro, and Peter Widmayer. 2011. A distributed algorithm for finding all best swap edges of a minimum-diameter spanning tree. *IEEE Trans. Depend. Secure Comput.* 8, 1 (2011), 1–12.

Thomas P. Hayes, Jared Saia, and Amitabh Trehan. 2012. The forgiving graph: A distributed data structure for low stretch under adversarial attack. *Distrib. Comput.* 25, 4 (2012), 261–278.

Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2013. Sublinear-time maintenance of breadth-first spanning tree in partially dynamic networks. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'13)*. 607–619.

Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2014b. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS'14)*. 146–155.

Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2014a. A subquadratic-time algorithm for dynamic single-source shortest paths. In *Proceedings of the Symposium on Discrete Algorithms (SODA'14)*. 1053–1072.

Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2016. Dynamic approximate all-pairs shortest paths: Breaking the $O(mn)$ barrier and derandomization. *SIAM J. Comput.* 45, 3 (2016), 947–1006.

Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Symposium on Theory of Computing (STOC'15)*. 21–30.

Giuseppe F. Italiano. 1991. Distributed algorithms for updating shortest paths. In *Proceedings of the International Workshop on Distributed Algorithms on Graphs (WDAG/DISC'91)*. 200–211.

Giuseppe F. Italiano and Rajiv Ramaswami. 1998. Maintaining spanning trees of small diameter. *Algorithmica* 22, 3 (1998), 275–304.

Hiro Ito, Kazuo Iwama, Yasuo Okabe, and Takuya Yoshihiro. 2005. Single backup table schemes for shortest-path routing. *Theor. Comput. Sci.* 333, 3 (2005), 347–353.

Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. 2012. Efficient distributed approximation algorithms via probabilistic tree embeddings. *Distrib. Comput.* 25, 3 (2012), 189–205.

Valerie King. 1999. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS'99)*. 81–91.

Liah Kor, Amos Korman, and David Peleg. 2013. Tight bounds for distributed minimum-weight spanning tree verification. *Theory Comput. Syst.* 53, 2 (2013), 318–340.

Amos Korman. 2008. Improved compact routing schemes for dynamic trees. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'08)*. 185–194.

Amos Korman and Shay Kutten. 2013. Controller and estimator for dynamic networks. *Info. Comput.* 223 (2013), 43–66.

Amos Korman and David Peleg. 2008. Dynamic routing schemes for graphs with low local density. *ACM Trans. Algor.* 4, 4 (2008), 41:1–41:18.

Danny Krizanc, Flaminia L. Luccio, and Rajeev Raman. 2004. Compact routing schemes for dynamic ring networks. *Theory Comput. Syst.* 37, 5 (2004), 585–607.

Shay Kutten and David Peleg. 1998. Fast distributed construction of small k-dominating sets and applications. *J. Algor.* 28, 1 (1998), 40–66.

Shay Kutten and Avner Porat. 1999. Maintenance of a spanning tree in dynamic networks. In *Proceedings of the International Symposium on Distributed Computing (DISC'99)*. 342–355.

Zvi Lotker, Boaz Patt-Shamir, and David Peleg. 2006. Distributed MST for constant diameter graphs. *Distrib. Comput.* 18, 6 (2006), 453–460.

Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. 2005. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM J. Comput.* 35, 1 (2005), 120–131.

Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Francisco, CA.

Navneet Malpani, Jennifer L. Welch, and Nitin H. Vaidya. 2000. Leader election algorithms for mobile ad hoc networks. In *Proceedings of the Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M'00)*. 96–103.

Enrico Nardelli, Guido Proietti, and Peter Widmayer. 2001. Finding all the best swaps of a minimum diameter spanning tree under transient edge failures. *J. Graph Algor. Appl.* 5, 5 (2001), 39–57.

Enrico Nardelli, Guido Proietti, and Peter Widmayer. 2003. Swapping a failing edge of a single source shortest paths tree is good and fast. *Algorithmica* 35, 1 (2003), 56–74.

David Peleg. 2000. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, Philadelphia, PA.

David Peleg and Vitaly Rubinovich. 2000. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.* 30, 5 (2000), 1427–1442.

K. V. S. Ramarao and S. Venkatesan. 1992. On finding and updating shortest paths distributively. *J. Algor.* 13, 2 (1992), 235–257.

Liam Roditty and Uri Zwick. 2011. On dynamic shortest paths problems. *Algorithmica* 61, 2 (2011), 389–401.

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

Received February 2015; revised March 2017; accepted August 2017