Deterministic Algorithms for Decremental Shortest Paths via Layered Core Decomposition*

Julia Chuzhoy† Thatchaphol Saranurak‡

Abstract

In the decremental single-source shortest paths (SSSP) problem, the input is an undirected graph $G = (V, E)$ with n vertices and m edges undergoing edge deletions, together with a fixed source vertex $s \in V$. The goal is to maintain a data structure that supports *shortest-path queries*: given a vertex $v \in V$, quickly return an (approximate) shortest path from s to v. The decremental all-pairs shortest paths (APSP) problem is defined similarly, but now the shortest-path queries are allowed between any pair of vertices of V.

Both problems have been studied extensively since the 80's, and algorithms with near-optimal total update time and query time have been discovered for them. Unfortunately, all these algorithms are randomized and, more importantly, they need to assume an *oblivious adversary* – a drawback that prevents them from being used as subroutines in several known algorithms for classical static problems. In this paper, we provide new *deterministic* algorithms for both problems, which, by definition, can handle an adaptive adversary.

Our first result is a deterministic algorithm for the decremental SSSP problem on *weighted* graphs with $O(n^{2+o(1)})$ total update time, that supports $(1+\epsilon)$-approximate shortest-path queries, with query time $O(|P| \cdot n^{o(1)})$, where P is the returned path. This is the first $(1+\epsilon)$-approximation adaptive-update algorithm supporting shortest-path queries in time below $O(n)$, that breaks the $O(mn)$ total update time bound of the classical algorithm of Even and Shiloah from 1981. Previously, Bernstein and Chechik [STOC'16, ICALP'17] provided a $\tilde{O}(n^2)$-time deterministic algorithm that supports approximate *distance* queries, but unfortunately the algorithm cannot return the approximate shortest paths. Chuzhoy and Khanna [STOC'19] showed an $O(n^{2+o(1)})$-time randomized algorithm for SSSP that supports approximate shortest-path queries in the adaptive adversary regime, but their algorithm only works in the restricted setting where only vertex deletions, and not edge deletions are allowed, and it requires $\Omega(n)$ time to respond to shortest-path queries.

Our second result is a deterministic algorithm for the decremental APSP problem on unweighted graphs that achieves total update time $O(n^{2.5+\delta})$, for any constant $\delta > 0$, supports approximate distance queries in $O(\log \log n)$ time, and supports approximate shortest-path queries in time $O(|E(P)| \cdot n^{o(1)})$, where P is the returned path; the algorithm achieves an $O(1)$-multiplicative and $n^{o(1)}$-additive approximation on the path length. All previous algorithms for APSP either assume an oblivious adversary or have an $\Omega(n^3)$ total update time when $m = \Omega(n^2)$, even if an $o(n)$-multiplicative approximation is allowed.

To obtain both our results, we improve and generalize the *layered core decomposition* data structure introduced by Chuzhoy and Khanna to be nearly optimal in terms of various parameters, and introduce a new generic approach of rooting Even-Shiloach trees at expander sub-graphs of the given graph. We believe both these technical tools to be interesting in their own right and anticipate them to be useful for designing future dynamic algorithms that work against an adaptive adversary.

1 Introduction

In the decremental single-source shortest path (SSSP) problem, the input is an undirected graph $G = (V, E)$ with n vertices and m edges undergoing edge deletions, together with a fixed source vertex $s \in V$. The goal is to maintain a data structure that supports *shortest-path* queries: given a vertex $v \in V$, quickly return an (approximate) shortest path from s to v. We also consider *distance* queries: given a vertex $v \in V$, return an approximate distance from s to v. The decremental all-pairs shortest path (APSP) problem is defined similarly, but now the shortest-path and distance queries are allowed between any pair $u, v \in V$ of vertices. A trivial algorithm for both problems is

*The full version of the paper can be accessed from https://arxiv.org/pdf/2009.08479.pdf

†Toyota Technological Institute at Chicago. Email: cjulia@ttic.edu. Part of the work was done while the author was a Weston visiting professor at the Department of Computer Science and Applied Mathematics, Weizmann Institute. Supported in part by NSF grant CCF-1616584.

‡Toyota Technological Institute at Chicago. Email: saranurak@ttic.edu.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

to simply maintain the current graph G, and, given a query between a pair u, v of vertices, run a BFS from one of these vertices, to report the shortest path between v and u in time $O(m)$. Our goal therefore is to design an algorithm whose *query time* – the time required to respond to a query – is significantly lower than this trivial $O(m)$ bound, while keeping the *total update time* – the time needed for maintaining the data structure over the entire sequence of updates, including the initialization — as small as possible. Observe that the best query time for shortest-path queries one can hope for is $O(|E(P)|)$, where P is the returned path[1].

Both SSSP and APSP are among the most well-studied dynamic graph problems. While almost optimal algorithms are known for both of them, all such algorithms are randomized and, more importantly, they assume an *oblivious adversary*. In other words, the sequence of edge deletions must be fixed in advance and cannot depend on the algorithm's responses to queries. Much of the recent work in the area of dynamic graphs has focused on developing so-called *adaptive-update algorithms*, that do not assume an oblivious adversary (see e.g. [NS17, WN17, NSW17, CGL+19] for dynamic connectivity, [BHI15, BHN16, BK19, Waj20] for dynamic matching, and [BC16, BC17, FHN16, Ber17, CK19, GWN20, BvdBG+20] for dynamic shortest paths); we also say that such algorithms work against an *adaptive adversary*. One of the motivating reasons to consider adaptive-update algorithms is that several algorithms for classical *static* problems need to use, as subroutines, dynamic graph algorithms that can handle adaptive adversaries (see e.g. [ST83, Mad10, CK19, CQ17]). In this paper, we provide new *deterministic* algorithms for both SSSP and APSP which, by definition, can handle adaptive adversary.

Throughout this paper, we use the $\tilde{O}$ notation to hide $\operatorname{poly}\log n$ factors, and $\widehat{O}$ notation to hide $n^{o(1)}$ factors, where n is the number of vertices in the input graph. We also assume that $\epsilon > 0$ is a small constant in the discussion below.

SSSP. The current understanding of decremental SSSP in the oblivious-adversary setting is almost complete, even for weighted graphs. Forster, Henzinger, and Nanongkai [FHN14a], improving upon the previous work of Bernstein and Roditty [BR11] and Forster et al. [FHN14b], provided a $(1+\epsilon)$-approximation algorithm, with close to the best possible total update time of $\widehat{O}(m \log L)$, where L is the ratio of largest to smallest edge length. The query time of the algorithm is also near optimal: approximate distance queries can be processed in $\tilde{O}(1)$ time, and approximate shortest-path queries in $\tilde{O}(|E(P)|)$ time, where P is the returned path. Due to known conditional lower bounds of $\widehat{\Omega}(mn)$ on the total update time for the exact version of SSSP[2], the guarantees provided by this algorithm are close to the best possible. Unfortunately, all these algorithms are randomized and need to assume an oblivious adversary.

For adaptive algorithms, the progress has been slower. It is well known that the classical algorithm of Even and Shiloach [ES81], that we refer to as ES-Tree throughout this paper, combined with the standard weight rounding technique (e.g. [Zwi98, Ber16]) gives a $(1+\epsilon)$-approximate deterministic algorithm for SSSP with $\tilde{O}(mn \log L)$ total update time and near-optimal query time. This bound was first improved by Bernstein [Ber17], generalizing a similar result of [BC16] for unweighted graphs, to $\tilde{O}(n^2 \log L)$ total update time. For the setting of sparse unweighted graphs, Bernstein and Chechik [BC17] designed an algorithm with total update time $\tilde{O}(n^{5/4}\sqrt{m}) \leq \tilde{O}(mn^{3/4})$, and Gutenberg and Wulff-Nielsen [GWN20] showed an algorithm with $\widehat{O}(m\sqrt{n})$ total update time.

Unfortunately, all of the above mentioned algorithms only support distance queries, but they cannot handle shortest-path queries. Recently, Chuzhoy and Khanna [CK19] attempted to fix this drawback, and obtained a randomized $(1+\epsilon)$-approximation *adaptive-update* algorithm with total expected update time $\widehat{O}(n^2 \log L)$, that supports shortest-path queries. Unfortunately, this algorithm has several other drawbacks. First, it is randomized. Second, the expected query time of $\tilde{O}(n \log L)$ may be much higher than the desired time that is proportional to the number of edges on the returned path. Lastly, and most importantly, the algorithm only works in the more restricted setting where only *vertex deletions* are allowed, as opposed to the more standard and general model with edge deletions[3]. Finally, a very recent work by Bernstein et

[1] Even though in extreme cases, where the graph is very sparse and the path P is very long, $O(|E(P)|)$ query time may be comparable to $O(m)$, for brevity, we will say that $O(|E(P)|)$ query time is below the $O(m)$ barrier, as is typically the case. For similar reasons, we will say that $O(|E(P)|)$ query time is below $O(n)$ query time.

[2] The bounds assume the Online Matrix-vector Multiplication (OMv) conjecture [FHNS15], and show that in order to achieve $O(n^{2-\delta})$ query time, for any constant $\delta > 0$, the total update time of $\Omega(n^{3-o(1)})$ is required in graphs with $m = \Theta(n^2)$.

[3] We emphasize that the vertex-decremental version is known to be strictly easier than the edge-decremental version for some problems. For example, there is a vertex-decremental algorithm for maintaining the exact distance between a fixed pair (s,t) of vertices in unweighted undirected graphs using $O(n^{2.932})$ total update time [San05] (later improved to $O(n^{2.724})$ in [vdBNS19]), but the edge-decremental version requires $\widehat{\Omega}(n^3)$ time when $m = \Omega(n^2)$ assuming the OMv conjecture [FHNS15]. A similar

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

al. [BvdBG+20], that is concurrent to this paper, shows $(1+\epsilon)$-approximate algorithms with $\widehat{O}(m\sqrt{n})$ total update time in unweighted graphs and $\tilde{O}(n^2 \log L)$ total update time in weighted graphs that can return an approximate shortest path P in $\tilde{O}(n)$ time (but not in time proportional to $|E(P)|$). The algorithm is randomized but works against an adaptive adversary.

As mentioned already, algorithms for approximate decremental SSSP are often used as subroutines in algorithms for static graph problems, including various flow and cut problems that we discuss below. Typically, in these applications, the following properties are desired from the algorithm for decremental SSSP:

- it should work against an adaptive adversary, and ideally it should be deterministic;
- it should be able to handle edge deletions (as opposed to only vertex deletions);
- it should support shortest-path queries, and not just distance queries; and
- it should have query time for shortest-path queries that is close to $O(|E(P)|)$, where P is the returned path.

In this paper we provide the first algorithm for decremental SSSP that satisfies all of the above requirements and improves upon the classical $\Omega(mn)$ bound of Even and Shiloach [ES81]. The total update time of the algorithm is $\widehat{O}(n^2 \log L)$, which is almost optimal for dense graphs.

Theorem 1.1 (Weighted SSSP) *There is a deterministic algorithm, that, given a simple undirected edge-weighted n-vertex graph G undergoing edge deletions, a source vertex s, and a parameter $\epsilon \in (1/n, 1)$, maintains a data structure in total update time $\widehat{O}(n^2(\frac{\log L}{\epsilon^2}))$, where L is the ratio of largest to smallest edge weights, and supports the following queries:*

- $\mathsf{dist\text{-}query}(s,v)$*: in $O(\log\log(nL))$ time return an estimate $\widetilde{\mathsf{dist}}(u,v)$, with $\mathsf{dist}_G(s,v) \le \widetilde{\mathsf{dist}}(s,v) \le (1+\epsilon)\mathsf{dist}_G(s,v)$; and*
- $\mathsf{path\text{-}query}(s,v)$*: either declare that s and v are not connected in G in $O(1)$ time, or return a s-v path P of length at most $(1+\epsilon)\mathsf{dist}_G(s,v)$, in time $\widehat{O}(|E(P)|\log\log L)$.*

Compared to the algorithm of [Ber17], our deterministic algorithm supports shortest-path, and not just distance queries, while having the same total update time up to a subpolynomial factor. Compared to the algorithm of [CK19], our algorithm handles the more general setting of edge deletions, is deterministic, and has faster query time. Compared to the work of [BvdBG+20] that is concurrent with this paper, our algorithm is deterministic and has a faster query time, though its total update time is somewhat slower for sparse graphs.

These improvements over previous works allow us to obtain faster algorithms for a number of classical static flow and cut problems; see the full version of the paper for more details. Most of the resulting algorithms are deterministic. For example, we obtain a deterministic algorithm for $(1+\epsilon)$-approximate minimum cost flow in unit edge-capacity graphs in $\widehat{O}(n^2)$ time. The previous algorithms by [LS14, AMV20] take time $\tilde{O}(\min\{m\sqrt{n}, m^{4/3}\})$, that is slower in dense graphs.

APSP. Our understanding of decremental APSP is also almost complete in the oblivious-adversary setting, even in weighted graphs. Bernstein [Ber16], improving upon the works of Baswana et al. [BHS07] and Roditty and Zwick [RZ12], obtained a $(1+\epsilon)$-approximation algorithm with $\tilde{O}(mn \log L)$ total update time, $O(1)$ query time for distance queries, and $\tilde{O}(|E(P)|)$ query time for shortest-path queries.[4] These bounds are conditionally optimal for small approximation factors[5]. Another line of work [BR11, FHN16, ACT14, FHN14a], focusing on larger approximation factors, recently culminated with a near-optimal result by Chechik [Che18]: for any integer $k \ge 1$, the algorithm of [Che18] provides a $((2+\epsilon)k-1)$-approximation, with $\widehat{O}(mn^{1/k}\log L)$ total update time and $O(\log\log(nL))$ query time for distance queries and $\tilde{O}(|E(P)|)$ query time for shortest-path queries. This result is near-optimal because all parameters almost match the best static algorithm of Thorup and Zwick [TZ01]. Unfortunately, both algorithms of Bernstein [Ber16] and of Chechik [Che18] need to assume an oblivious adversary.

In contrast, our current understanding of adaptive-update algorithms is very poor even for unweighted graphs. The classical ES-Tree algorithm of Even and Shiloach [ES81] implies a deterministic algorithm for decremental exact APSP in unweighted graphs with $O(mn^2)$ total update time and optimal query time of $O(|E(P)|)$ where P is the returned path. This running time was first improved by Forster, Henzinger, and Nanongkai [FHN16], who showed a deterministic $(1+\epsilon)$-

separation holds for decremental exact APSP.

[4] Bernstein's algorithm works even in directed graphs.

[5] Assuming the BMM conjecture [DHZ00, RZ11] or the OMv conjecture [FHNS15], 1.99-approximation algorithms for decremental APSP require $\widehat{\Omega}(n^3)$ total update time or $\widehat{\Omega}(n)$ query time in undirected unweighted graphs when $m = \Omega(n^2)$.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

approximation algorithm with $\tilde{O}(mn)$ total update time and $O(\log \log n)$ query time for distance queries. Recently, Gutenberg and Wulff-Nilsen [GWN20] significantly simplified this algorithm. Despite a long line of research, the state-of-the-art in terms of total update time remains $\tilde{O}(mn)$, which can be as large as $\tilde{\Theta}(n^3)$ in dense graphs, in any algorithm whose query time is below the $O(n)$ bound. To highlight our lack of understanding of the problem, no adaptive algorithms that attain an $o(n^3)$ total update time and query time below $O(n)$ for shortest-path queries are currently known for any density regime, even if we allow huge approximation factors, such as, for example, any $o(n)$-approximation[6].

In this work, we break this barrier by providing the first **deterministic** algorithm with **sub-cubic** total update time, that achieves a **constant** multiplicative and a **subpolynomial** additive approximation:

Theorem 1.2 (Unweighted APSP) *There is a deterministic algorithm, that, given a simple unweighted undirected n-vertex graph G undergoing edge deletions and a parameter $1 \leq k \leq o(\log^{1/8} n)$, maintains a data structure using total update time of $\widehat{O}(n^{2.5+2/k})$ and supports the following queries:*

- $\mathsf{dist\text{-}query}(u, v)$*: in $O(\log n \log \log n)$ time return an estimate $\widetilde{\mathsf{dist}}(u, v)$, where $\mathsf{dist}_G(u, v) \leq \widetilde{\mathsf{dist}}(u, v) \leq 3 \cdot 2^k \cdot \mathsf{dist}_G(u, v) + \widehat{O}(1)$; and*
- $\mathsf{path\text{-}query}(u, v)$*: either declare that u and v are not connected in $O(\log n)$ time, or return a u-v path P of length at most $3 \cdot 2^k \cdot \mathsf{dist}_G(u, v) + \widehat{O}(1)$, in $\widehat{O}(|E(P)|)$ time.*

The additive approximation term in dist-query *and* path-query *is* $\exp(O(k \log^{3/4} n)) = \widehat{O}(1)$.

For example, by letting k be a large enough constant, we can obtain a total update time of $\widehat{O}(n^{2.501})$, constant multiplicative approximation, and $\exp(O(\log^{3/4} n))$ additive approximation.

We note that the concurrent work of [BvdBG+20] on dynamic spanners that was mentioned above implies a randomized $\tilde{O}(1)$-multiplicative approximation adaptive-update algorithm for APSP with $\tilde{O}(m)$ total update time but it requires a large $\tilde{O}(n)$ query time even for distance queries; in contrast, our algorithm is deterministic and has faster query times: $\widehat{O}(|E(P)|)$ for shortest-path and $O(\log n \log \log n)$ for distance queries.

[6]When we allow a factor-n approximation, one can use deterministic decremental connectivity algorithms (e.g. [HdLT01]) with $\tilde{O}(m)$ total update time and $O(\log n)$ query time for distance queries.

Technical Highlights. Both our algorithms for SSSP and APSP are based on the *Layered Core Decomposition* (LCD) data structure introduced by Chuzhoy and Khanna [CK19]. Informally, one may think of the data structure as maintaining a "compressed" version of the graph. Specifically, it maintains a decomposition of the current graph G into a relatively small number of expanders (called cores), such that every vertex of G either lies in one of the cores, or has a short path connecting it to one of the cores. The data structure supports approximate shortest-path queries within the cores, and queries that return, for every vertex of G, a short path connecting it to one of the cores. Chuzhoy and Khanna [CK19] presented a randomized algorithm for maintaining the LCD data structure, as the graph G undergoes **vertex** deletions, with total update time $\widehat{O}(n^2)$. As our first main technical contribution, we improve and generalize their algorithm in a number of ways: first, our algorithm is deterministic; second, it can handle the more general setting of edge deletions and not just vertex deletions; we improve the total update time to the near optimal bound of $\widehat{O}(m)$; and we improve the query times of this algorithm. We further motivate this data structure and discuss the technical barriers that we needed to overcome in order to obtain these improvements in Section 3. We believe that the LCD data structure is of independent interest and will be useful in future adaptive-update dynamic algorithms. Indeed, a near-optimal *short-path oracle on decremental expanders*, which is one of the technical ingredients of our LCD data structure, has already found further applications in other algorithms for dynamic problems [BGS20].

Our second main contribution is a new generic method to exploit the Even-Shiloach tree (ES-Tree) data structure[7]. Many previous algorithms for SSSP and APSP [BR11, FHN14a, FHN16, Che18] need to maintain a collection $\mathcal{T}$ of several ES-Trees. One drawback of this approach, is that, whenever the root of an ES-Tree is disconnected due to a sequence of edge deletions, we need to reinitialize a new ES-Tree, leading to high total update time. To overcome this difficulty, most such algorithms choose the locations of the roots of the trees *at random*, so that they are "hidden" from an oblivious adversary, and hence cannot be disconnected too often. Clearly, this approach fails completely against an adaptive adversary, that can repeatedly delete edges incident to the roots of the trees.

In order to withstand an adaptive adversary, we introduce the idea of "rooting an ES-Tree at an ex-

[7]Here, we generally include variants such as the monotone ES-Tree.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

pander" instead. As an expander is known to be robust against edge deletions even from an adaptive adversary [NS17, NSW17, SW19], the adversary cannot disconnect the "expander root" of the tree too often, leading to smaller total update time. The LCD data structure naturally allows us to apply this high level idea, as it maintains a relatively small number of expander subgraphs (cores). This leads to our algorithm for APSP in the small distance regime. We also use this idea to implement the short-path oracle on expanders. We believe that our general approach of "rooting a tree at an expander" instead of "rooting a tree at a random location" will be a key technique for future adaptive-update algorithms. This idea was already exploited in a different way in a recent subsequent work [BGS20].

Organization. We provide preliminaries in Section 2. Section 3 focuses on our main technical contribution: the new LCD data structure. In this extended abstract, we only describe the overview of this data structure. The details of its implementation, which is the main technical contribution of this paper, can be found in the full version of the paper. We exploit this data structure to obtain our algorithms for SSSP and APSP in Section 4 and Section 5, respectively. The new algorithms for cut/flow problems that rely on our algorithm for SSSP via known reductions appear in the full version of the paper.

2 Preliminaries

All graphs considered in this paper are undirected and simple, so they do not have parallel edges or self loops. Given a graph G and a vertex $v \in V(G)$, we denote by $\deg_G(v)$ the degree of v in G. Given a length function $\ell : E(G) \to \mathbb{R}$ on the edges of G, for a pair u, v of vertices in G, we denote by $\mathsf{dist}_G(u, v)$ the length of the shortest path connecting u to v in G, with respect to the edge lengths $\ell(e)$. As the graph G undergoes edge deletions, the notation $\deg_G(v)$ and $\mathsf{dist}_G(u, v)$ always refer to the current graph G. For a path P in G, we denote $|P| = |E(P)|$.

Given a graph G and a subset S of its vertices, let $G[S]$ be the subgraph of G induced by S. We denote by $\delta_G(S)$ the total number of edges of G with exactly one endpoint in set S, and we let $E_G(S)$ be the set of all edges of G with both endpoints in S. Given two subsets A, B of vertices of G, we let $E_G(A, B)$ denote the set of all edges with one endpoint in A and another in B. The *volume* of a vertex set S is $\mathrm{vol}_G(S) = \sum_{v \in S} \deg_G(v)$. If S is a set of vertices with $1 \leq |S| < |V(G)|$, then we may refer to S as a *cut*, and we denote $\overline{S} = V(G) \setminus S$. We let the *conductance* of the cut S be $\Phi_G(S) = \frac{\delta_G(S)}{\min\{\mathrm{vol}_G(S), \mathrm{vol}_G(\overline{S})\}}$. We may omit the subscript G when clear from context. We denote $\mathrm{vol}(G) = \sum_{v \in V(G)} \deg_G(v) = 2|E(G)|$. Given a graph G, we let its conductance $\Phi(G)$ be the minimum, over all cuts S, of $\Phi_G(S)$. Notice that $0 \leq \Phi(G) \leq 1$ always holds. We say that graph G is a *φ-expander* iff $\Phi(G) \geq \varphi$.

Given a graph G, its *k-orientation* is an assignment of a direction to each undirected edge of G, so that each vertex of G has out-degree at most k. For a given orientation of the edges, for each vertex $u \in V(G)$, we denote by $\text{in-deg}_G(u)$ and $\text{out-deg}_G(u)$ the in-degree and out-degree of u, respectively. Note that, if G has a k-orientation, then for every subset $S \subseteq V$ of its vertices, $|E_G(S)| \leq k \cdot |S|$ must hold, and, in particular, $|E(G)| \leq k \cdot |V(G)|$. We say that a set $F \subseteq E(G)$ of edges has a k-orientation if the graph induced by F has a k-orientation.

Decremental Connectivity/Spanning Forest. We use the results of [HdLT01], who provide a deterministic data structure, that we denote by $\mathsf{CONN\text{-}SF}(G)$, that, given an n-vertex unweighted undirected graph G, that is subject to edge deletions, maintains a spanning forest of G, with total update time $O((m+n)\log^2 n)$, where m is the number of edges in the initial graph G. Moreover, the data structure supports connectivity queries $\mathsf{conn}(G, u, v)$: given a pair u, v of vertices of G, return "yes" if u and v are connected in G, and "no" otherwise. The running time to respond to each such query is $O(\log n / \log\log n)$.

Even-Shiloach Trees. Suppose we are given a graph $G = (V, E)$ with integral lengths $\ell(e) \geq 1$ on its edges $e \in E$, a source s, and a distance bound $D \geq 1$. Even-Shiloach Tree (ES-Tree) algorithm maintains a shortest-path tree from vertex s, that includes all vertices v with $\mathsf{dist}(s, v) \leq D$, and, for every vertex v with $\mathsf{dist}(s, v) \leq D$, the distance $\mathsf{dist}(s, v)$. Typically, ES-Tree only supports edge deletions (see, e.g. [ES81, Din06, HK95]). However, as shown in [BC16, Lemma 2.4], it is easy to extend the data structure to also handle edge insertions in the following two cases: either (i) at least one of the endpoints of the inserted edge is a singleton vertex, or (ii) the distances from the source s to other vertices do not decrease due to the insertion. We denote the corresponding data structure from [BC16] by $\mathsf{ES\text{-}Tree}(G, s, D)$. It was shown in [BC16] that the total update time of $\mathsf{ES\text{-}Tree}(G, s, D)$, including the initialization and all edge deletions, is $O(mD + U)$, where U is the total number of updates (edge insertions or deletions), and m is the total number of edges that ever appear in G.

Greedy Degree Pruning. We consider a simple degree pruning procedure defined in [CK19]. Given a graph H and a degree bound d, the procedure computes

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

a vertex set $A \subseteq V(H)$, as follows. Start with $A = V(H)$. While there is a vertex $v \in A$, such that fewer than d neighbors of v lie in A, remove v from A. We denote this procedure by Proc-Degree-Pruning(H, d) and denote by A the output of the procedure. The following observation was implicitly shown in [CK19]; for completeness, we provide its proof in the full version of the paper.

Observation 2.1 *Let A be the outcome of procedure* Proc-Degree-Pruning(H, d)*, for any graph H and integer d. Then A is the* unique maximal *vertex set such that every vertex in $H[A]$ has degree at least d. That is, for any subset A' of $V(H)$ where $H[A']$ has minimum degree at least d, $A' \subseteq A$ must hold.*

Consider now a graph H that undergoes edge deletions, and let A denote the outcome of procedure Proc-Degree-Pruning(H, d) when applied to the current graph. Notice that, from the above observation, set A is a *decremental vertex set*, that is, vertices can only leave the set, as edges are deleted from H. We use the following algorithm, that we call Alg-Maintain-Pruned-Set(H, d), that allows us to maintain the set A as the graph H undergoes edge deletions; the algorithm is implicit in [CK19].

The algorithm Alg-Maintain-Pruned-Set(H, d) starts by running Proc-Degree-Pruning(H, d) on the original graph H. Recall that the procedure initializes $A = V(H)$, and then iteratively deletes from A vertices v that have fewer than d neighbors in A. In the remainder of the algorithm, we simply maintain the degree of every vertex in $H[A]$ as H undergoes edge deletions. Whenever, for any vertex v, $\deg_{H[A]}(v)$ falls below d, we remove v from A. Observe that vertex degrees in $H[A]$ are monotonically decreasing. Moreover, each degree decrement at a vertex v can be charged to an edge that is incident to v and was deleted from $H[A]$. As each edge is charged at most twice, the total update time is $O(|E(H)|+|V(H)|)$. Therefore, we obtain the following immediate observation.

Observation 2.2 *The total update time of* Alg-Maintain-Pruned-Set *is $O(m + |V(H)|)$, where m is the number of edges that belonged to graph H at the beginning. Moreover, whenever the algorithm removes some vertex v from set A, vertex v has fewer than d neighbors in A in the current graph H.*

3 Layered Core Decomposition

Our main technical contribution is a data structure called *Layered Core Decomposition* (LCD), that improves and generalizes the data structure introduced in [CK19]. In order to define the data structure, we need to introduce the notions of *virtual vertex degrees*, and a partition of vertices into *layers*, which we do next.

Suppose we are given an n-vertex m-edge graph $G = (V, E)$ and a parameter $\Delta > 1$. We emphasize that throughout this section, the input graph G is unweighted, and the length of a path P in G is the number of its edges. Let $d_{\max}$ be the largest vertex degree in G. Let r be the smallest integer, such that $\Delta^{r-1} > d_{\max}$. Note that $r \leq O(\log_\Delta n)$. Next, we define degree thresholds $h_1, h_2, \ldots, h_r$, as follows: $h_j = \Delta^{r-j}$. Therefore, $h_1 > d_{\max}$, $h_r = 1$, and for all $1 < j \leq r$, $h_j = h_{j-1}/\Delta$. For convenience, we also denote $h_{r+1} = 0$.

Definition. (Virtual Vertex Degrees and Layers) *For all $1 \leq j \leq r$, let A_j be the outcome of* Proc-Degree-Pruning(G, h_j)*, when applied to the current graph G. The* virtual degree $\widetilde{\deg}(v)$ *of v in G is the largest value h_j such that $v \in A_j$. If no such value exists, then $\widetilde{\deg}(v) = h_{r+1} = 0$. For all $1 \leq j \leq r+1$, let $\Lambda_j = \{v \mid \widetilde{\deg}(v) = h_j\}$ denote the set of vertices whose virtual degree is h_j. We call Λ_j the j*th layer.

Note that for every vertex $v \in V(G)$, $\widetilde{\deg}(v) \in \{h_1, \ldots, h_{r+1}\}$. Also, $\Lambda_1 = \emptyset$ since all vertex degrees are below h_1, and Λ_{r+1}, the set of vertices with virtual degree 0, contains all isolated vertices. For all $1 \leq j' < j \leq r+1$, we say that layer $\Lambda_{j'}$ is *above* layer Λ_j. For convenience, we write $\Lambda_{\leq j} = \bigcup_{j'=1}^{j} \Lambda_{j'}$ and $\Lambda_{<j}, \Lambda_{\geq j}, \Lambda_{>j}$ are defined similarly. Notice that $\Lambda_{\leq j} = A_j$. For any vertex u, let $\deg_{\leq j}(u) = |E_G(u, \Lambda_{\leq j})|$ denote the number of neighbors of u that lie in layer j or above.

Intuitively, the partition of $V(G)$ into layers is useful because, in a sense, we can tightly control the degrees of vertices in each layer. This is summarized more formally in the following three observations. The first observation, that follows immediately from Observation 2.1, shows that every vertex in layer Λ_j has many neighbors in layer j and above:

Observation 3.1 *Throughout the algorithm, for each $1 \leq j \leq r+1$, for each vertex $u \in \Lambda_j$, $\deg_{\leq j}(u) \geq h_j$. Therefore, the minimum vertex degree in $G[\Lambda_{\leq j}]$ is always at least h_j.*

As observed already, from Observation 2.1, over the course of the algorithm, vertices may only be deleted from $\Lambda_{\leq j} = A_j$. This immediately implies the following observation:

Observation 3.2 *As edges are deleted from G, for every vertex v, $\widetilde{\deg}(v)$ may only decrease.*

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Throughout, we denote by $n_{\leq j}$ the number of vertices that belonged to $\Lambda_{\leq j}$ at the beginning of the algorithm, before any edges were deleted from the input graph. Observe that $n_{\leq j}h_j \leq 2m$ by Observation 3.1. The proof of the following observation appears in the full version of the paper.

Observation 3.3 *For all $1 \leq j \leq r$, let $E_{\geq j}$ be the set of all edges, such that at any point of time at least one endpoint of e lied in $\Lambda_{\geq j}$. Then $E_{\geq j}$ has a (Δh_j)-orientation, and so $|E_{\geq j}| \leq \Delta h_j n$. Moreover, the total number of edges e, such that, at any point of the algorithm's execution, both endpoints of e lied in Λ_j, is bounded by $n_{\leq j}h_j\Delta$.*

From Observation 3.1, all vertex degrees in $G[\Lambda_{\leq j}]$ are at least h_j, so, in a sense, graph $G[\Lambda_{\leq j}]$ is a high-degree graph. One advantage of high-degree graphs is that every pair of vertices lying in the same connected component of such a graph must have a short path connecting them; specifically, it is not hard to show that, if u, v are two vertices lying in the same connected component C of graph $G[\Lambda_{\leq j}]$, then there is a path connecting them in C, of length at most $O(|V(C)|/h_j)$. This property of graphs $G[\Lambda_{\leq j}]$ is crucial to our algorithms for SSSP and APSP, and one of the goals of the LCD data structure is to support *short-path* queries: given a pair of vertices $u, v \in \Lambda_{\leq j}$, either report that they lie in different connected components of $G[\Lambda_{\leq j}]$, or return a path of length at most roughly $O(|V(C)|/h_j)$ connecting them, where C is the connected component of $G[\Lambda_{\leq j]}$ containing u and v. Additionally, one can show that a high-degree graph must contain a *core decomposition*. Specifically, suppose we are given a simple n-vertex graph H, with minimum vertex degree at least h. Intuitively, a *core* of H is a vertex-induced sub-graph $K \subseteq H$, such that, for $\varphi = \Omega(1/\log n)$, graph K is a φ-expander, and all vertex degrees in K are at least $\varphi h/3$. One can show that, if K is a core, then its diameter is $O(\log n/\varphi)$, and it is $(\varphi h/3)$-edge-connected. A *core decomposition* of H is a collection $\mathcal{F} = \{K_1, \ldots, K_t\}$ of vertex-disjoint cores, such that, for each vertex $u \notin \bigcup_{K\in\mathcal{F}} V(K)$, there are at least $2h/3$ edge-disjoint paths of length $O(\log n)$ from u to vertices in $\bigcup_{K\in\mathcal{F}} V(K)$. The results of [CK19] implicitly show the existence of a core decomposition in a high-degree graph, albeit with a much more complicated definition of the cores and of the decomposition. For completeness, in the full version of the paper, we formally state and prove a theorem about the existence of a core decomposition in a high-degree graph. Though we do not need this theorem for the results of this paper, we feel that it is an interesting graph theoretic statement in its own right, that in a way motivates the LCD data structure, whose intuitive goal is to maintain a layered analogue of the core decomposition of the input graph G, as it undergoes edge deletions.

Formally, the LCD data structure receives as input an (unweighted) graph G undergoing edge deletions, and two parameters $\Delta \geq 2$ and $1 \leq q \leq o(\log^{1/4} n)$. It maintains the partition of $V(G)$ into layers $\Lambda_1, \ldots, \Lambda_{r+1}$, as described above, and additionally, for each layer Λ_j, the data structure maintains a collection $\mathcal{F}_j$ of vertex-disjoint subgraphs of the graph $H_j = G[\Lambda_j]$, called *cores* (while we do not formally have any requirements from the cores, e.g. we do not formally require that a core is an expander, our algorithm will in fact still ensure that this is the case, so the intuitive description of the cores given above matches what our algorithm actually does). Throughout, we use an additional parameter $\gamma(n) = \exp(O(\log^{3/4} n)) = \widehat{O}(1)$. The data structure is required to support the following three types of queries:

- Short-Path(j, u, v): Given any pair of vertices u and v from $\Lambda_{\leq j}$, either report that u and v lie in different connected components of $G[\Lambda_{\leq j}]$, or return a simple path P connecting u to v in $G[\Lambda_{\leq j}]$ of length $O(|V(C)|(\gamma(n))^{O(q)}/h_j) = \widehat{O}(|V(C)|/h_j)$, where C is the connected component of $G[\Lambda_{\leq j}]$ containing u and v.
- To-Core-Path(u): Given any vertex u, return a simple path $P = (u = u_1, \ldots, u_z = v)$ of length $O(\log^3 n)$ from u to a vertex v that lies in some core in $\bigcup_j \mathcal{F}_j$. Moreover, path P must visit the layers in a non-decreasing order, that is, if $u_i \in \Lambda_j$ then $u_{i+1} \in \Lambda_{\leq j}$.
- Short-Core-Path(K, u, v): Given any pair of vertices u and v, both of which lie in some core $K \in \bigcup_j \mathcal{F}_j$, return a simple u-v path P in K of length at most $(\gamma(n))^{O(q)} = \widehat{O}(1)$.

We now formally state one of our main technical results - an algorithm for maintaining the LCD data structure under edge deletions.

Theorem 3.1 (Layered Core Decomposition) *There is a deterministic algorithm that, given a simple unweighted n-vertex m-edge graph $G = (V, E)$ undergoing edge deletions, and parameters $\Delta \geq 2$ and $1 \leq q \leq o(\log^{1/4} n)$, maintains a partition $(\Lambda_1, \ldots, \Lambda_{r+1})$ of V into layers, where for all $1 \leq j \leq r+1$, each vertex in Λ_j has virtual degree h_j. Additionally, for each layer Λ_j, the algorithm maintains a collection $\mathcal{F}_j$ of vertex-disjoint subgraphs of the graph $H_j = G[\Lambda_j]$, called* cores. *The algorithm supports queries* Short-Path(j, u, v) *in time $O(\log n)$ if u and v*

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

lie in different connected components of $G[\Lambda_{\leq j}]$*, and in time* $O(|P|(\gamma(n))^{O(q)}) = \widehat{O}(|P|)$ *otherwise, where* P *is the* u*-*v *path returned. Additionally, it supports queries* To-Core-Path(u) *with query time* $O(|P|)$*, where* P *is the returned path, and* Short-Core-Path(K, u, v) *with query time* $(\gamma(n))^{O(q)} = \widehat{O}(1)$*. For all* $1 \leq j \leq h+1$*, once a core* K *is added to* $\mathcal{F}_j$ *for the first time, it only undergoes edge- and vertex-deletions, until* $K = \emptyset$ *holds. The total number of cores ever added to* $\mathcal{F}_j$ *throughout the algorithm is at most* $\widehat{O}(n\Delta/h_j)$*. The total update time of the algorithm is* $\widehat{O}(m^{1+1/q}\Delta^{2+1/q}(\gamma(n))^{O(q)}) = \widehat{O}(m^{1+1/q}\Delta^{2+1/q})$.

For intuition, it is convenient to set the parameters $\Delta = 2$ and $q = \log^{1/8} n$, which is also the setting that we use in algorithms for SSSP and for APSP in the large-distance regime. For this setting, $(\gamma(n))^{O(q)} = \widehat{O}(1)$, and the total update time of the algorithm is $\widehat{O}(m)$.

Optimality. The guarantees of the LCD data structure from Theorem 3.1 are close to optimal in several respects. First, the total update time of $\widehat{O}(m)$ and the query time for Short-Core-Path and To-Core-Path are clearly optimal to within a subpolynomial in n factor. The length of the path returned by Short-Path queries is almost optimal in the sense that there can exist a path P in a connected component C of $G[\Lambda_{\leq j}]$ whose length is $\Omega(|V(C)|/h_j)$; the query time of $\widehat{O}(|P|)$ is almost optimal as well. The bound on the total number of cores ever created in Λ_j is also near optimal. This is because, even in the static setting, there exist graphs with minimum degree h_j that require $\tilde{\Omega}(n/h_j)$ cores in order to guarantee the desired properties of a core decomposition.

Comparison with the Algorithm of [CK19] and Summary of Main Challenges. The data structure from [CK19] supports the same set of queries, but has several significant drawbacks compared to the results of Theorem 3.1. First, the algorithm of [CK19] is randomized. Moreover, it can only handle vertex deletions, as opposed to the more general and classical setting of edge deletions (which is also required in some applications to static flow and cut problems). Additionally, the total update time of the algorithm of [CK19] is $\widehat{O}(n^2)$, as opposed to the almost linear running time of $\widehat{O}(m)$ of our algorithm. For every index j, the total number of cores ever created in Λ_j can be as large as $\widehat{O}(n^2/h_j^2)$ in the algorithm of [CK19], as opposed to the bound of $\widehat{O}(n/h_j)$ that we obtain; this bound directly affects the running of our algorithm for APSP. Lastly, the query time for Short-Path(j, u, v) is only guaranteed to be bounded by $\widehat{O}(|V(C)|)$ in [CK19], where C is a connected component of $\Lambda_{\leq j}$ to which u and v belong, as opposed to our query time of $\widehat{O}(|P|)$, where P is the u-v path returned. This faster query time is essential in order to obtain the desired query time of $\widehat{O}(|P|)$ in our algorithms for SSSP and APSP. Next, we describe some of the challenges to achieving these improvements, and also sketch some ideas that allowed us to overcome them.

Vertex deletions versus edge deletions. The algorithm of [CK19] maintains, for every index $1 \leq j \leq r$, a variation of the core decomposition (that is based on vertex expansion) in graph $G[\Lambda_j]$. This decomposition can be computed in almost linear time $\widehat{O}(|E(\Lambda_j)|) = \widehat{O}(nh_j)$, which is close to the best time one can hope for, creating an initial set $\mathcal{F}_j$ of at most $\widehat{O}(n/h_j)$ cores. Since every core $K \in \mathcal{F}_j$ has vertex degrees at least $h_j/n^{o(1)}$, the decomposition can withstand up to $h_j/(2n^{o(1)})$ vertex deletions, while maintaining all its crucial properties. However, after $h_j/(2n^{o(1)})$ vertex deletions, some cores may become disconnected, and the core decomposition structure may no longer retain the desired properties. Therefore, after every batch of roughly $h_j/(2n^{o(1)})$ vertex deletions, the algorithm of [CK19] recomputes the core decomposition $\mathcal{F}_j$ from scratch. Since there may be at most n vertex-deletion operations throughout the algorithm, the core decomposition $\mathcal{F}_j$ only needs to be recomputed at most $\widehat{O}(n/h_j)$ times throughout the algorithm, leading to the total update time of $\widehat{O}(n/h_j) \cdot \widehat{O}(|E(\Lambda_j)|) = \widehat{O}(n^2)$. The total number of cores that are ever added to $\mathcal{F}_j$ over the course of the algorithm is then bounded by $\widehat{O}(n/h_j) \cdot \widehat{O}(n/h_j) = \widehat{O}(n^2/h_j^2)$.

Consider now the edge-deletion setting. Even if we are willing to allow a total update time of $\widehat{O}(n^2)$, we cannot hope to perform a single computation of the decomposition $\mathcal{F}_j$ in time faster than linear in $|E(\Lambda_j)|$, that is, $O(nh_j)$. Therefore, we can only afford at most $O(n/h_j)$ such re-computations over the course of the algorithm. Since the total number of edges in graph $G[\Lambda_j]$ may be as large as $\Theta(nh_j)$, our core decomposition must be able to withstand up to h_j^2 edge deletions. However, even after just h_j edge deletions, some vertices of Λ_j may become disconnected in graph $G[\Lambda_{\leq j}]$, and some of the cores may become disconnected as well. In order to overcome his difficulty, we first observe that it takes $h_j/n^{o(1)}$ edge deletions before a vertex in Λ_j becomes "useless", which roughly means that it is not well-connected to other vertices in Λ_j. Similarly to the algorithm of [CK19], we would now like to recompute the core decomposition $\mathcal{F}_j$ only after $h_j/(2n^{o(1)})$ vertices of Λ_j become useless, which roughly corresponds to $h_j^2/n^{o(1)}$ edge deletions. Additionally, we

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

employ the expander pruning technique from [SW19] in order to maintain the cores so that they can withstand this significant number of edge deletions. As in [CK19], this approach can lead to $\widehat{O}(n^2)$ total update time, ensuring that the total number of cores that are ever added to set $\mathcal{F}_j$ is at most $\widehat{O}(n^2/h_j^2)$.

Obtaining faster total update time and fewer cores. Even with the modifications described above, the resulting total update time is only $\widehat{O}(n^2)$, while our desired update time is near-linear in m. It is not hard to see that recomputing the whole decomposition $\mathcal{F}_j$ from scratch every time is too expensive, and with the $\widehat{O}(m)$ total update time we may only afford to do so at most $\widehat{O}(1)$ times. In order to overcome this difficulty, we further partition each layer Λ_j into *sublayers* $\Lambda_{j,1}, \Lambda_{j,2}, \ldots, \Lambda_{j,L_j}$ whose sizes are geometrically decreasing (that is, $|\Lambda_{j,\ell}| \approx |\Lambda_{j,\ell-1}|/2$ for all ℓ). The core decompositions $\mathcal{F}_{j,\ell}$ will be computed in each sub-layer separately, and the final core decomposition for layer j that the algorithm maintains is $\mathcal{F}_j = \bigcup_\ell \mathcal{F}_{j,\ell}$. In general, we guarantee that, for each ℓ, $|\Lambda_{j,\ell}| \leq n_{\leq j}/2^{\ell-1}$ always holds, and we recompute the core decomposition $\mathcal{F}_{j,\ell}$ for sublayer at $\Lambda_{j,\ell}$ at most $\widehat{O}(2^\ell)$ times. We use Observation 3.3 to show that $|E(\Lambda_{j,\ell})| \leq h_j\Delta \cdot n_{\leq j}/2^{\ell-1} = O(m/2^\ell)$ must hold. Therefore, the total time for computing core decompositions within each sublayer is $\widehat{O}(m)$. As there are $O(\log n)$ sublayers within a layer, the total time for computing the decompositions over all layers is $\widehat{O}(m)$. This general idea is quite challenging to carry out, since, in contrast to layers $\Lambda_1, \ldots, \Lambda_{r+1}$, where vertices may only move from higher to lower layers throughout the algorithm, the vertices of a single layer can move between its sublayers in a non-monotone fashion. One of the main challenges in the design of the algorithm is to design a mechanism for allowing the vertices to move between the sublayers, so that the number of such moves is relatively small.

Improving query times. The algorithm of [CK19] supports Short-Core-Path(K, u, v) queries, that need to return a short path inside the core K connecting the pair u, v of its vertices, in time $\tilde{O}(|V(K)|) + \widehat{O}(1)$, returning a path of length $\widehat{O}(1)$; in contrast our algorithm takes time $\widehat{O}(1)$. The query time of Short-Core-Path(K, u, v) in turn directly influences the query time of Short-Path(u, v) queries, which in turn is critical to the final query time that we obtain for SSSP and APSP problems. Another way to view the problem of supporting Short-Core-Path(K, u, v) queries is the following: suppose we are given an expander graph K that undergoes edge- and vertex-deletions (in batches). We are guaranteed that after each batch of such updates, the remaining graph K is still an expander, and so every pair of vertices in K has a path of length $O(\text{poly}\log n)$ connecting them. The goal is to support "short-path" queries: given a pair u, v of vertices of K, return a path of length $\widehat{O}(1)$ connecting them. The problem seems interesting in its own right, and, for example, it plays an important role in the recent fast deterministic approximation algorithm for the sparsest cut problem of [CGL+19]. The algorithm of [CK19], in order to process Short-Core-Path(K, u, v) query, simply perform a breadth-first search in the core K to find the required u-v path, leading to the high query time. Instead, we develop a more efficient algorithm for supporting short-path queries in expander graphs, that is similar in spirit and in techniques to the algorithm of [CGL+19]. This new data structure has already found further applications to other problems [BGS20].

For Short-Path(u, v) queries, the guarantees of [CK19] are similar to our guarantees in terms of the length of the path returned, but their query processing time is too high, and may be as large as $\tilde{\Omega}(n)$ in the worst case. We improve the query time to $\widehat{O}(|P|)$, where P is the returned path, which is close to the best possible bound. This improvement is necessary in order to obtain faster algorithms for several applications to cut and flow problems that we discuss. The improvement is achieved by exploiting the improved data structure that supports Short-Core-Path queries within the cores, and by employing a *minimum* spanning tree data structure on top of the core decomposition, instead of using dynamic connectivity as in the algorithm of [CK19].

Randomized versus Deterministic Algorithm. While the algorithm of [CK19] works against an adaptive adversary, it is a randomized algorithm. The two main randomized components of the algorithm are: (i) an algorithm to compute a core decomposition; and (ii) data structure that supports Short-Core-Path(K, u, v) queries within each core. For the first component, we exploit the recent fast deterministic algorithm for the Balanced Cut problem of [CGL+19]. For the second component, as discussed above, we design a new deterministic algorithm that support Short-Core-Path(K, u, v) queries within the cores. These changes lead to a deterministic algorithm for the LCD data structure.

Using the LCD Data Structure for SSSP and APSP With our improved implementation of the LCD data structure, using the same approach as that of [CK19], we immediately obtain the desired algorithm for SSSP, proving Theorem 1.1.

Our algorithm for APSP in the large-distance regime exploits the LCD data structure in a way similar

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

to that of the algorithm for SSSP: We use the LCD data structure in order to "compress" the dense parts of the graph. In the sparse part, instead of maintaining a single ES-Tree, as in the algorithm for SSSP, we maintain the deterministic tree cover of [GWN20] (which simplifies the moving ES-Tree data structure of [FHN16]).

Our algorithm for APSP in the small-distance regime uses a *tree cover* approach, similar to previous work [BR11, FHN14a, FHN16, Che18]. The key difference is that we root each ES-Tree at one of the cores maintained by the LCD data structure (recall that each core is a high-degree expander), instead of rooting it at a random vertex.

The proof of Theorem 3.1, which is the key technical contribution of this paper can be found in the full version of the paper. However, the statement of this theorem is sufficient in order to obtain our results for SSSP and APSP, that are discussed in Section 4 and Section 5, respectively.

4 SSSP

This section is dedicated to the proof of Theorem 1.1. The main idea is identical to that of [CK19], who use the framework of [Ber17], combined with a weaker version of the LCD data structure. The improvements in the guarantees that we obtain follow immediately by plugging the new LCD data structure from Section 3 into their algorithm. As is the standard practice in such algorithms, we treat each distance scale separately. We prove the following theorem that allows us to handle a single distance scale.

Theorem 4.1 *There is a deterministic algorithm, that, given a simple undirected n-vertex graph G with weights on edges that undergoes edge deletions, together with a source vertex $s \in V(G)$ and parameters $\epsilon \in (1/n, 1)$ and $D > 0$, supports the following queries:*

- $\mathsf{dist\text{-}query}_D(s,v)$*: in time $O(1)$, either correctly report that $\mathsf{dist}_G(s,v) > 2D$, or return an estimate $\widetilde{\mathsf{dist}}(s,v)$. Moreover, if $D \leq \mathsf{dist}_G(s,v) \leq 2D$, then $\mathsf{dist}_G(s,v) \leq \widetilde{\mathsf{dist}}(s,v) \leq (1+\epsilon)\mathsf{dist}_G(s,v)$ must hold.*

- $\mathsf{path\text{-}query}_D(s,v)$*: either correctly report that $\mathsf{dist}_G(s,v) > 2D$ in time $O(1)$, or return a s-v path P in time $\widehat{O}(|P|)$. Moreover, if $D \leq \mathsf{dist}_G(s,v) \leq 2D$, then the length of P must be bounded by $(1+\epsilon)\mathsf{dist}_G(s,v)$. Path P may not be simple, but an edge may appear at most once on P.*

  The total update time of the algorithm is $\widehat{O}(n^2/\epsilon^2)$.

We provide a proof of Theorem 4.1 below, after we complete the proof of Theorem 1.1 using it, via standard arguments.

We will sometimes refer to edge weights as edge lengths, and we denote the length of an edge $e \in E(G)$ by $\ell(e)$. We assume that the minimum edge weight is 1 by scaling, so the maximum edge weight is L. For all $0 \leq i \leq \lceil \log(Ln) \rceil$, we maintain a data structure from Theorem 4.1 with the distance parameter $D_i = 2^i$. Therefore, the total update time of our algorithm is bounded by $\widehat{O}(n^2(\frac{\log L}{\epsilon^2}))$, as required.

In order to respond to a query $\mathsf{dist\text{-}query}(s,v)$, we perform a binary search on the values D_i, and run queries $\mathsf{dist\text{-}query}_{D_i}(s,v)$ in the corresponding data structure. Clearly, we only need to perform at most $O(\log\log(Ln))$ such queries, in order to respond to query $\mathsf{dist\text{-}query}(s,v)$.

In order to respond to $\mathsf{path\text{-}query}(s,v)$, we first run the algorithm for $\mathsf{dist\text{-}query}(s,v)$ in order to identify a distance scale D_i, for which $D_i \leq \mathsf{dist}_G(s,v) \leq 2D_i$ holds. We then run query $\mathsf{path\text{-}query}_{D_i}(s,v)$ in the corresponding data structure.

In order to complete the proof of Theorem 1.1, it now remains to prove Theorem 4.1, which we do in the remainder of this section.

Recall that we have denoted by $\ell(e)$ the length/weight of the edge e of G. We use standard edge-weight rounding to show that we can assume that $D = \lceil 4n/\epsilon \rceil$ and that all edge lengths are integers between 1 and $4D$. In order to achieve this, we discard all edges whose length is greater than $2D$, and we set the length of each remaining edge e to be $\ell'(e) = \lceil 4n\ell(e)/(\epsilon D) \rceil$. For every pair u, v of vertices, let $\mathsf{dist}'(u,v)$ denote the distance between u and v with respect to the new edge length values. Notice that for all u, v, $\frac{4n}{\epsilon D}\mathsf{dist}(u,v) \leq \mathsf{dist}'(u,v) \leq \frac{4n}{\epsilon D}\mathsf{dist}(u,v) + n$, since the shortest s-v path contains at most n edges. Moreover, if $\mathsf{dist}(u,v) \geq D$, then $n \leq \mathsf{dist}(u,v) \cdot \frac{n}{D}$, so $\mathsf{dist}'(u,v) \leq \frac{4n}{\epsilon D}\mathsf{dist}(u,v) + \frac{n}{D}\mathsf{dist}(u,v) \leq \frac{4n}{\epsilon D}\mathsf{dist}(u,v)(1+\epsilon/4)$. Notice also that, if $D \leq \mathsf{dist}(u,v) \leq 2D$, then $\lceil \frac{4n}{\epsilon} \rceil \leq \mathsf{dist}'(u,v) \leq 4\lceil \frac{4n}{\epsilon} \rceil$. Therefore, from now on we can assume that $D = \lceil 4n/\epsilon \rceil$, and for simplicity, we will denote the new edge lengths by $\ell(e)$ and the corresponding distances between vertices by $\mathsf{dist}(u,v)$. From the above discussion, all edge lengths are integers between 1 and $4D$. It is now enough to prove Theorem 4.1 for this setting, provided that we ensure that, whenever $D \leq \mathsf{dist}(s,v) < 4D$ holds, we return a path of length at most $(1+\epsilon/2)\mathsf{dist}(s,v)$ in response to query $\mathsf{path\text{-}query}(v)$.

The Algorithm. Let m denote the initial number of edges in the input graph G. We partition all edges of G into $\lambda = \lfloor \log(4D) \rfloor$ classes, where for $0 \leq i \leq \lambda$, edge e belongs to *class* i iff $2^i \leq \ell(e) < 2^{i+1}$. We denote

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

the set of all edges of G that belong to class i by E^i. Fix an index $1 \leq i \leq \lambda$, and let G_i be the sub-graph of G induced by the edges in E^i. We view G_i as an unweighted graph and maintain the LCD data structure from Theorem 3.1 on G_i with parameter $\Delta = 2$ and $q = \log^{1/8} n$ using total update time $\widehat{O}(m^{1+1/q}\Delta^{2+1/q}) = \widehat{O}(m)$. Recall that $\gamma(n) = \exp(O(\log^{3/4} n))$.

We let $\alpha = (\gamma(n))^{O(q)} = \widehat{O}(1)$ be chosen such that, in response to query Short-Path(j, u, v), the LCD data structure must return a path of length at most $|V(C)| \cdot \alpha/h_j$, where C denotes the connected component of graph $G[\Lambda_{\leq j}]$ containing u and v. We use the parameter $\tau_i = \frac{8n\lambda\alpha}{\epsilon D} \cdot 2^i$ that is associated with graph G_i. This parameter is used to partition the vertices of G into a set of vertices that are *heavy* with respect to class i, and vertices that are *light* with respect of class i. Specifically, we let $U_i = \left\{v \in V(G_i) \mid \widetilde{\deg}_{G_i}(v) \geq \tau_i\right\}$ be the set of vertices that are heavy for class i, and we let $\overline{U}_i = V(G_i) \setminus U_i$ be the set of vertices that are light for class i.

Next, we define the heavy and the light graph for class i. The *heavy graph for class i*, that is denoted by G_i^H, is defined as $G_i[U_i]$. In other words, its vertex set is the set of all vertices that are heavy for class i, and its edge set is the set of all class-i vertices whose both endpoints are heavy for class i. The *light graph for class i*, denoted by G_i^L, is defined as follows. Its vertex set is $V(G_i)$, and its edge set contains all edges $e \in E_i$, such that at least one endpoint of e lies in $\overline{U}_i$. Notice that we can exploit the LCD data structure to compute the initial graphs G_i^H and G_i^L, and to maintain them, as edges are deleted from G.

Our algorithm exploits the LCD data structure in two ways. First, observe that, from Observation 3.3, for all $1 \leq i \leq \lambda$, the total number of edges that ever belong to the light graph G_i^L over the course of the algorithm is bounded by $O(n\tau_i)$. Additionally, we will exploit the Short-Path queries that the LCD data structure supports.

Let j_i be the largest integer, such that $h_{j_i} \geq \tau_i$ (recall that h_j is the virtual degree of vertices in layer Λ_j). Given a query Short-Path(j_i, u, v) to the LCD data structure on G_i, where u and v lie in the same connected component C of G_i^H, the data structure must return a simple u-v path in C, containing at most $\frac{|V(C)|\alpha}{\tau_i}$ edges. Abusing the notation, we denote this query by Short-Path(C, u, v) instead.

Let $G^L = \bigcup_{i=1}^{\lambda} G_i^L$ be the *light graph* for the graph G. Next, we define an *extended light graph* $\hat{G}^L$, as follows. We start with $\hat{G}^L = G^L$; the vertices of G^L are called *regular vertices*. Next, for every $1 \leq i \leq \lambda$, for every connected component C of G_i^H, we add a vertex v_C to $\hat{G}^L$, that we call a *special vertex*, or a *supernode*, and connect it to every regular vertex $u \in V(C)$ with an edge of length $1/4$.

For all $1 \leq i \leq \lambda$, we use the CONN-SF data structure on graph G_i^H, in order to maintain its connected components. The total update time of these connectivity data structures is bounded by $O(m\lambda) \leq O(m \log D)$.[8] The following observation follows immediately from the assumption that all edge lengths in G are at least 1.

Observation 4.1 *Throughout the algorithm, for every vertex $v \in V(G)$, $\mathsf{dist}_{\hat{G}^L}(s, v) \leq \mathsf{dist}_G(s, v)$.*

The following theorem was proved in [CK19]; the proof follows the arguments from [Ber17] almost exactly.

Theorem 4.2 (Theorem 4.4 in [CK19]) *There is a deterministic algorithm, that maintains an approximate single-source shortest-path tree T of graph $\hat{G}^L$ from the source s, up to distance $8D$. Tree T is a sub-graph of $\hat{G}^L$, and for every vertex $v \in V(\hat{G}^L)$, with $\mathsf{dist}_{\hat{G}^L}(s, v) \leq 8D$, the distance from s to v in T is at most $(1 + \epsilon/4)\mathsf{dist}_{\hat{G}^L}(s, v)$. The total update time of the algorithm is $\tilde{O}\left(\frac{nD}{\epsilon} + |E(G)| + \sum_{e \in E} \frac{D}{\epsilon \ell(e)}\right)$, where $E(G)$ is the set of edges that belong to G at the beginning of the algorithm, and E is the set of all edges that are ever present in the graph $\hat{G}^L$.*

Recall that $D = \Theta(n/\epsilon)$. Since, for all $1 \leq i \leq \lambda$, the total number of edges of E^i ever present in $\hat{G}^L$ is bounded by $O(n\tau_i) = O\left(n \cdot \frac{8n\lambda\alpha}{\epsilon D} \cdot 2^i\right) = \widehat{O}(n \cdot 2^i)$ from Observation 3.3, and since the total number of edges incident to the special vertices that are ever present in $\hat{G}^L$ is bounded by $O(n\lambda \log n) = \tilde{O}(n)$, we get that the running time of the algorithm from Theorem 4.2 is bounded by:

$$\tilde{O}\left(\frac{n^2}{\epsilon^2} + \sum_{i=1}^{\lambda} \frac{|E^i|D}{\epsilon \cdot 2^i}\right) = \widehat{O}\left(\frac{n^2}{\epsilon^2}\right).$$

As other components take $\widehat{O}(m)$ time, the total update time of the algorithm for Theorem 4.1 is $\widehat{O}(n^2/\epsilon^2)$, as required. It remains to show how the algorithm responds to queries path-query$_D(s, v)$ and dist-query$_D(s, v)$.

[8] We note that our setting is slightly different from that of [Ber17], who used actual vertex degrees and not their virtual degrees in the definitions of the light and the heavy graphs. Our definition is identical to that of [CK19], though they did not define the virtual degrees explicitly. However, they used Procedure Proc-Degree-Pruning in order to define the heavy and the light graphs, and so their definition of both graphs is identical to ours, except for the specific choice of the thresholds τ_i).

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Responding to $\mathsf{path\text{-}query}_D(s,v)$**.** Given a query $\mathsf{path\text{-}query}_D(s,v)$, we start by computing the unique simple s-v path P in the tree T given by Theorem 4.2. If vertex v is not in T, then clearly $\mathsf{dist}_G(s,v) > 2D$ and so we report $\mathsf{dist}_G(s,v) > 2D$. From now, we assume $v \in T$. Next, we transform the path P in $\hat{G}^L$ into an s-v path P^* in the original graph G as follows.

Let $v_{C_1}, \ldots, v_{C_z}$ be all special vertices that appear on the path P. For $1 \leq k \leq z$, let u_k be the regular vertex preceding v_{C_k} on P, and let u'_k be the regular vertex following v_{C_k} on P. If C_k is a connected component of a heavy graph G_i^H of class i, we use the query $\mathsf{Short\text{-}Path}(C_k, u_k, u'_k)$ in the LCD data structure for graph G_i in order to to obtain a simple u_k-u'_k path Q_k contained in C_k, that contains at most $\frac{|V(C_k)|\alpha}{\tau_i}$ (unweighted) edges. Then, we replace the vertex v_{C_k} with the path Q_k on path P. As we can find the path P in time $O(|P|)$, by following the tree T, and since the query time to compute each path Q_k is bounded by $|Q_k| \cdot (\gamma(n))^{O(q)} = \widehat{O}(|Q_k|)$, the total time to compute path P^* is bounded by $\widehat{O}(|E(P^*)|)$.

We now bound the length of the path P^*. Recall that, by Observation 4.1, path P has length $(1+\epsilon/4)\mathsf{dist}_{\hat{G}^L}(s,v) \leq (1+\epsilon/4)\mathsf{dist}_G(s,v)$. For each $1 \leq i \leq \lambda$, let $\mathcal{C}_i = \{C_k \mid v_{C_j} \in P \text{ and } C_k \text{ is a connected component of } G_i^H\}$. Let $\mathcal{Q}_i$ be the set of all corresponding paths Q_k of $C_k \in \mathcal{C}_i$. We can bound the total length of all path in $\mathcal{Q}_i$ as follows:

$$\begin{aligned}\sum_{Q \in \mathcal{Q}_i} \ell(Q) &\leq \sum_{C_k \in \mathcal{C}_i} |Q_k| \cdot 2^{i+1} \\ &\leq \sum_{C_k \in \mathcal{C}_i} \frac{|V(C_k)|\alpha}{\tau_i} \cdot 2^{i+1} \\ &\leq \sum_{C_k \in \mathcal{C}_i} |V(C_k)| \cdot \frac{\epsilon D}{4n\lambda} \\ &\leq \frac{\epsilon D}{4\lambda}\end{aligned}$$

(we have used the fact that $\tau_i = \frac{8n\lambda\alpha}{\epsilon D} \cdot 2^i$, and that all components in $\mathcal{C}_i$ are vertex-disjoint). Summing up over all λ classes, the total length of all paths Q_k corresponding to the special vertices on path P is at most $\epsilon D/4$. We conclude that $\ell(P^*) \leq \ell(P) + \epsilon D/4$. If $\mathsf{dist}_G(s,v) \geq D$, we have that $\ell(P^*) \leq (1+\epsilon/4)\mathsf{dist}_G(s,v) + \epsilon\mathsf{dist}_G(s,v)/4 = (1+\epsilon/2)\mathsf{dist}_G(s,v)$. Notice that path P^* may not be simple, since a vertex may belong to several heavy graphs G_i^H. However, for every edge $e \in E(G)$, there is a unique index i such that $e \in G_i$, and the sets of edges of the heavy graph G_i^H and the light graph G_i^L are disjoint from each other. In particular, if $e \in E(G_i^H)$, then $e \notin \hat{G}^L$. Since path P is simple, all graphs $C_1, \ldots, C_z$ are edge-disjoint from each other, and their edges are also disjoint from $E(\hat{G}^L)$. We conclude that an edge may appear at most once on P^*.

Responding to $\mathsf{dist\text{-}query}_D(s,v)$**.** Given a query $\mathsf{dist\text{-}query}_D(s,v)$, we simply return $\mathsf{dist}'(s,v) = \mathsf{dist}_T(s,v) + \epsilon D/4$ in time $O(1)$. Recall that $\mathsf{dist}'(s,v) = \mathsf{dist}_T(s,v) + \epsilon D/4 \geq \ell(P^*) \geq \mathsf{dist}_G(s,v)$ (here, P^* is the path that we would have returned in response to query $\mathsf{path\text{-}query}_D(s,v)$, though we only use this path for the analysis and do not compute it explicly). As before if $\mathsf{dist}_G(s,v) \geq D$, then, from Observation 4.1, $\mathsf{dist}'(s,v) \leq (1+\epsilon/2)\mathsf{dist}_G(s,v)$.

5 APSP

In this section, we prove Theorem 1.2 by combining two algorithms. We use the function $\gamma(n) = \exp(O(\log^{3/4} n))$ from Theorem 3.1.

The first algorithm, summarized in the next theorem, is faster in the large-distance regime:

Theorem 5.1 (APSP for large distances) *There is a deterministic algorithm, that, given parameters $0 < \epsilon < 1/2$ and $D > 0$, and a simple unweighted undirected n-vertex graph G that undergoes edge deletions, maintains a data structure using total update time of $\widehat{O}\left(n^3/(\epsilon^3 D)\right)$ and supports the following queries:*

- $\mathsf{dist\text{-}query}_D(u,v)$*: either correctly declare that $\mathsf{dist}_G(u,v) > 2D$ in $O(\log n)$ time, or return an estimate $\mathsf{dist}'(u,v)$ in $O(\log n)$ time. If $D \leq \mathsf{dist}_G(u,v) \leq 2D$, then $\mathsf{dist}_G(u,v) \leq \mathsf{dist}'(u,v) \leq (1+\epsilon)\mathsf{dist}_G(u,v)$ must hold.*
- $\mathsf{path\text{-}query}_D(u,v)$*: either correctly declare that $\mathsf{dist}_G(u,v) > 2D$ in $O(\log n)$ time, or return a u-v path P of length at most $9D$ in $\widehat{O}(|P|)$ time. If $D \leq \mathsf{dist}_G(u,v) \leq 2D$, then $|P| \leq (1+\epsilon)\mathsf{dist}_G(u,v)$ must hold.*

The second algorithm is faster for the short-distance regime.

Theorem 5.2 (APSP for small distances) *There is a deterministic algorithm, that, given parameters $1 \leq k < o(\log^{1/8} n)$ and $D > 0$, and a simple unweighted undirected n-vertex graph G that undergoes edge deletions, maintains a data structure using total update time $\widehat{O}(n^{2+3/k}D)$ and supports the following queries:*

- $\mathsf{dist\text{-}query}_D(u,v)$*: in time $O(1)$, either correctly establish that $\mathsf{dist}_G(u,v) > 2D$, or correctly establish that $\mathsf{dist}(u,v) \leq 2^k \cdot 3D + (\gamma(n))^{O(k)}$.*
- $\mathsf{path\text{-}query}_D(u,v)$*: either correctly establish that $\mathsf{dist}_G(u,v) > 2D$ in $O(1)$ time, or return a u-v path*

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

P of length at most $2^k \cdot 3D + (\gamma(n))^{O(k)}$, in time $O(|P|) + (\gamma(n))^{O(k)}$.

We prove Theorems 5.1 and 5.2 below, after we complete the proof of Theorem 1.2 using them. Let $\epsilon = 1/4$, and $D^* = n^{0.5-1/k}$. For $1 \leq i \leq \left\lceil \log_{1+\epsilon} n \right\rceil$, let $D_i = (1+\epsilon)^i$. For all $1 \leq i \leq \left\lceil \log_{1+\epsilon} n \right\rceil$, if $D_i \leq D^*$, then we maintain the data structure from Theorem 5.2 with the value $D = D_i$, and the input parameter k, and otherwise we maintain the data structure from Theorem 5.1 with the bound $D = D_i$ and the parameter ϵ. Since, from the statement of Theorem 1.2, $k \leq o(\log^{1/8} n)$ holds, it is easy to verify that the total update time for maintaining these data structures is bounded by $\widehat{O}(n^{2.5+2/k})$.

Given a query $\mathsf{dist\text{-}query}(u, v)$, we perform a binary search on indices i, in order to find an index for which $\mathsf{dist}_G(u,v) > 2D_i$ and $\mathsf{dist}_G(u,v) < 2^k \cdot 3D_{i+1} + (\gamma(n))^{O(k)}$ hold, by querying the data structures from Theorems 5.2 and 5.1. We then return $\widetilde{\mathsf{dist}}(u,v) = 2^k \cdot 3 \cdot D_{i+1} + (\gamma(n))^{O(k)}$ as a response to the query. Notice that we are guaranteed that $\widetilde{\mathsf{dist}}(u,v) \leq 2^k \cdot 3 \cdot \mathsf{dist}_G(u,v) + \widehat{O}(1)$, as required. As there are $O(\log n)$ possible values of D_i, the query time is $O(\log n \log\log n)$.

Given a query $\mathsf{path\text{-}query}(u, v)$, we start by checking whether u and v are connected, for example by running $\mathsf{dist\text{-}query}_D(u, v)$ query with $D = (1+\epsilon)n$ on the data structure from Theorem 5.1. If u and v are not connected, then we can report this in time $O(\log n)$. Otherwise, we perform a binary search on indices i exactly as before, to find an index for which $\mathsf{dist}_G(u,v) > 2D_i$ and $\mathsf{dist}_G(u,v) < 2^k \cdot 3D_{i+1} + (\gamma(n))^{O(k)}$ hold. Then, we use query in the appropriate data structure, $\mathsf{path\text{-}query}_{D_{i+1}}(u, v)$ and obtain a u-v path P of length at most $2^k \cdot 3D_{i+1} + (\gamma(n))^{O(k)} \leq 2^k \cdot 3 \cdot \mathsf{dist}_G(u,v) + \widehat{O}(1)$, in time $\widehat{O}(|P|)$.

5.1 The Large-Distance Regime

The goal of this section is to prove Theorem 5.1. The algorithm easily follows by combining our algorithm for SSSP with the algorithm of [GWN20] for APSP (that simplifies the algorithm of [FHN16] for the same problem).

Data Structures and Update Time Our starting point is an observation of [GWN20], that we can assume w.l.o.g. that throughout the edge deletion sequence, the graph G remains connected. Specifically, we will maintain a graph G^*, starting with $G^* = G$. Whenever an edge e is deleted from G, as part of the input update sequence, if the removal of e does not disconnect the graph G, then we delete e from G^* as well. Otherwise, we ignore this edge deletion operation, and edge e remains in G^*. It is easy to see that in the latter case, edge e is a bridge in G^*, and will remain so until the end of the algorithm. It is also immediate to verify that, if u, v are two vertices that lie in the same connected component of G, then $\mathsf{dist}_G(u,v) = \mathsf{dist}_{G^*}(u,v)$. Moreover, if P is any (not necessarily simple) path connecting u to v in graph G^*, such that an edge may appear at most once on P, then P is also a u-v path in graph G.

Throughout the algorithm, we use two parameters: $R^c = \epsilon D/8$ and $R^d = 4D$. We maintain the following data structures.

- Data structure $\mathsf{CONN\text{-}SF}(G)$ for dynamic connectivity. Recall that the data structure has total update time $\tilde{O}(m)$, and it supports connectivity queries $\mathsf{conn}(G, u, v)$: given a pair u, v of vertices of G, return "yes" if u and v are connected in G, and "no" otherwise. The running time to respond to each such query is $O(\log n / \log\log n)$.
- A collection $S \subseteq V(G)$ of *source vertices*, with $|S| \leq O(n/R^c) \leq O(n/(\epsilon D))$;
- For every source vertex $s \in S$, the data structure from Theorem 4.1, in graph G^*, with source vertex s, distance bound R^d, and accuracy parameter $\epsilon = 1/4$.

Recall that the data structure from Theorem 4.1 has total update time $\widehat{O}(n^2/\epsilon^2)$. Since we will maintain $O(n/(\epsilon D))$ such data structures, the total update time for maintaining them is $\widehat{O}(n^3/(\epsilon^3 D))$.

Consider now some source vertex $s \in S$, and the data structure from Theorem 4.1 that we maintain for it. Since graph G is unweighted, all edges of G belong to a single class, and so the algorithm will only maintain a single heavy graph (instead of maintaining a separate heavy graph for every edge class), and a single light graph. In particular, this ensures that at any time during the algorithm's execution, all cores in $\bigcup_j \mathcal{F}_j$ are vertex-disjoint. In order to simplify the notation, we denote the extended light graph that is associated with graph G^* by $\hat{G}^L$; recall that this graph does not depend on the choice of the vertex s. Recall that, from Observation 4.1, throughout the algorithm, for every vertex $v \in V(G^*)$, $\mathsf{dist}_{\hat{G}^L}(s,v) \leq \mathsf{dist}_{G^*}(s,v)$ holds. Additionally, the data structure maintains an ES-Tree, that we denote by $\tau(s)$, in graph $\hat{G}^L$, that is rooted at the vertex s, and has depth R^d. We say that the source s *covers* a vertex $v \in V(G)$ iff the distance from v to s in the tree $\tau(s)$ is at most R^c.

Our algorithm will maintain, together with each vertex $v \in V(G)$, a list of all source vertices $s \in S$ that cover v, together with a pointer to the location

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

of v in the tree $\tau(s)$. We also maintain a list of all source vertices $s' \in S$ with $v \in \tau(s')$, together with a pointer to the location of v in $\tau(s')$. These data structures can be easily maintained along with the trees $\tau(s)$ for $s \in S$. The total update time for maintaining the ES-Trees subsumes the additional required update time.

We now describe an algorithm for maintaining the set S of source vertices. We start with $S = \emptyset$. Throughout the algorithm, vertices may only be added to S, but they may never be deleted from S. At the beginning, before any edge is deleted from G, we initialize the data structure as follows. As long as some vertex $v \in V(G)$ is not covered by any source, we select any such vertex v, add it to the set S of source vertices, and initialize the data structure $\tau(v)$ for the new source vertex v. This initialization algorithm terminates once every vertex of G is covered by some source vertex in S. As edges are deleted from G and distances between vertices increase, it is possible that some vertex $v \in V(G)$ stops being covered by vertices of S. Whenever this happens, we add such a vertex v to the set S of source vertices, and initialize the corresponding data structure $\tau(v)$. We need the following claim.

Claim 5.1 *Throughout the algorithm, $|S| \leq O(n/R^c)$ holds.*

Proof. For a source vertex $s \in S$, let $C(s)$ be the set of all vertices at distance at most $R^c/2$ from vertex s in graph $\hat{G}^L$. From the algorithm's description, and since the distances between regular vertices in the graph $\hat{G}^L$ may only grow over the course of the algorithm, for every pair $s, s' \in S$ of source vertices, $\mathsf{dist}_{\hat{G}^L}(s, s') \geq R^c$ holds throughout the algorithm, and so $C(s) \cap C(s') = \emptyset$. Since graph G^* is a connected graph throughout the algorithm, so is graph $\hat{G}^L$. It is then easy to verify that, if $|S| \geq 2$, then for every source vertex $s \in S$, $|C(s)| \geq \Omega(|R^c|)$ (we have used the fact that graph G is unweighted, and so, in graph $\hat{G}^L$, all edges have lengths in $\{1/4, 1\}$). It follows that $|S| \leq O(n/R^c)$. □

Responding to $\mathsf{path\text{-}query}_D(x, y)$ queries. Suppose we are given a query $\mathsf{path\text{-}query}_D(x, y)$, where x, y are two vertices of G. Recall that our goal is to either correctly establishes that $\mathsf{dist}_G(x, y) > 2D$, or to return an x-y path P in G, of length at most $9D$. We also need to ensure that, if $D \leq \mathsf{dist}_G(x, y) \leq 2D$, then $|P| \leq (1 + \epsilon)\mathsf{dist}_G(x, y)$.

Our first step is to use query $\mathsf{conn}(G, x, y)$ in data structure $\mathsf{CONN\text{-}SF}(G)$ in order to check whether x and y lie in the same connected component of G. If this is not the case then we report that x and y are not connected in G. Therefore, we assume from now on that x and y are connected in G. Recall that the running time for query $\mathsf{conn}(G, x, y)$ is $O(\log n / \log \log n)$.

Recall that our algorithm ensures that there is some source vertex $s \in S$ that covers x. Therefore, $\mathsf{dist}_{\hat{G}_L}(s, x) \leq R^c$. It is also easy to verify that $\mathsf{dist}_{\hat{G}^L}(x, y) \leq \mathsf{dist}_{G^*}(x, y)$ must hold. Therefore, if $\mathsf{dist}_G(x, y) \leq 2D$, $y \in \tau(s)$ must hold. We can find the source vertex s that covers x and check whether $y \in \tau(s)$ in time $O(1)$ using the data structures that we maintain. If $y \notin \tau(s)$, then we are guaranteed that $\mathsf{dist}_G(x, y) > 2D$. We terminate the algorithm and report this fact.

Therefore, we assume from now on that $y \in \tau(s)$. We compute the unique simple x-y path P in the tree $\tau(s)$, by retracing the tree from x and y until we find their lowest common ancestor; this can be done in time $O(|P|)$. The remainder of the algorithm is similar to that for responding to queries for the SSSP data structure. We denote by $v_{C_1}, \ldots, v_{C_z}$ the sequence of all special vertices that appear on the path P. For $1 \leq k \leq z$, let u_k be the regular vertex preceding v_{C_k} on P, and let u'_k be the regular vertex following v_{C_k} on P. We then use queries $\mathsf{Short\text{-}Path}(C_k, u_k, u'_k)$ to the LCD data structure in order to obtain a simple u_k-u'_k path Q_k contained in C_k. Then, we replace the vertex v_{C_k} with the path Q_k on path P. As in the analysis of the algorithm for SSSP, the running time of this algorithm is bounded by $\widehat{O}(|E(P^*)|)$, and the length of the path P^* is bounded by $|P| + \epsilon R^d \leq \mathsf{dist}_G(x, y) + 4\epsilon D$. Since $|P| \leq 2R^d \leq 8D$, this is bounded by $9D$. Moreover, if $D \leq \mathsf{dist}_G(x, y) \leq 2D$, then we are guaranteed that the length of P^* is at most $(1 + 4\epsilon)\mathsf{dist}_G(x, y)$. The running time of the algorithm is $O(\log n)$ if it declares that $\mathsf{dist}_G(x, y) > 2D$, and it is bounded by $\widehat{O}(|P^*|)$ if a path P^* is returned. We note that every edge may appear at most once on path P^*. Indeed, an edge of G^* may belong to the heavy graph, or to the extended light graph $\hat{G}^L$, but not both of them. Therefore, an edge of P may not lie on any of the paths in $\{Q_1, \ldots, Q_z\}$. Moreover, since path P is simple, the connected components $C_1, \ldots, C_k$ of the heavy graph are all disjoint, and so the paths $Q_1, \ldots, Q_z$ must he disjoint from each other. Therefore, every edge may appear at most once on path P^*. As observed before, this means that P^* is contained in the graph G.

Responding to $\mathsf{dist\text{-}query}_D(x, y)$. The algorithm is similar to that for $\mathsf{path\text{-}query}_D(x, y)$. As before, our first step is to use query $\mathsf{conn}(G, x, y)$ in data structure $\mathsf{CONN\text{-}SF}(G)$ in order to check whether x and y lie in the same connected component of G. If this is not the case then we report that x and y are not connected in G. Therefore, we assume from now on that x and y are connected in G. Recall that the running time for query

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

$\mathsf{conn}(G, x, y)$ is $O(\log n / \log\log n)$.

As before, we find a source s that covers vertex x, and check whether $y \in \tau(s)$, in time $O(1)$. If this is not the case, then we correctly report that $\mathsf{dist}_G(x, y) > 2D$, and terminate the algorithm. Otherwise, we return an estimate $\mathsf{dist}'(x, y) = \mathsf{dist}_{\hat{G}^L}(x, s) + \mathsf{dist}_{\hat{G}^L}(y, s) + 4\epsilon D$. This can be done in time $O(1)$, by reading the distance labels of x and y in tree $\tau(s)$. From the above arguments, we are guaranteed that there is an x-y path P^* in G, whose length is at most $\mathsf{dist}'(x, y)$, so $\mathsf{dist}_G(x, y) \leq \mathsf{dist}'(x, y)$ must hold. Notice that $\mathsf{dist}_{\hat{G}^L}(y, s) \leq \mathsf{dist}_{\hat{G}^L}(x, s) + \mathsf{dist}_{\hat{G}^L}(x, y) \leq R^c + \mathsf{dist}_G(x, y)$. Therefore, $\mathsf{dist}'(x, y) \leq 2R^c + 4\epsilon D + \mathsf{dist}_G(x, y) \leq 8\epsilon D + \mathsf{dist}_G(x, y)$. Therefore, if $\mathsf{dist}_G(x, y) \geq D$, then $\mathsf{dist}'(x, y) \leq (1 + 8\epsilon)\mathsf{dist}_G(x, y)$ must hold.

In order to obtain the guarantees required in Theorem 5.1, we use the parameter $\epsilon' = \epsilon/8$, and run the algorithm described above while using ϵ' instead of ϵ. It is easy to verify that the resulting algorithm provides the desired guarantees.

5.2 The Small-Distance Regime In this section, we prove Theorem 5.2. Recall that we are given a simple unweighted graph G undergoing edge deletions, a parameter $k \geq 1$ and a distance scale D. We set $\Delta = n^{1/k}$ and $q = 10k$.

Our data structure is based on the LCD data structure from Theorem 3.1. We invoke the algorithm from Theorem 3.1 on the input graph G, with parameters Δ and q. Recall that the algorithm maintains a partition of the vertices of G into layers $\Lambda_1, \ldots, \Lambda_{r+1}$, and notice that $r \leq k + 1$. Let $\alpha = (\gamma(n))^{O(q)}$ be chosen such that, in response to the Short-Core-Path and To-Core-Path queries, the length of the path returned by the LCD data structure is guaranteed to be at most α. For every index $1 < j \leq r$, we define two distance parameters: R^d_j called a *distance radius* and R^c_j called a *covering radius* as follows:

$$R^d_j = 2^{r-j}(3D + 2\alpha k) \text{ and } R^c_j = R^d_j - 2D.$$

Note that $R^d_j \leq 2^{k-1} \cdot 3D + 2^k \alpha k = O(D \cdot (\gamma(n))^{O(k)})$ for all $j > 1$. (As $\Lambda_1 = \emptyset$, we only give the bound for all $j > 1$). Recall that the LCD data structure maintains a collection $\mathcal{F}_j$ of cores for each level $j > 1$. We need the following key concept:

Definition. *A vertex $v \in \Lambda_j$ is a* far vertex *iff $\mathsf{dist}_G(v, \Lambda_{<j}) > R^d_j$. A core $K \in \mathcal{F}_j$ is a* far core *iff all vertices in K are far vertices, that is, $\mathsf{dist}_G(V(K), \Lambda_{<j}) > R^d_j$.*

Observe that once a core K becomes a far core, it remains a far core, until it is destroyed. This is because distances in G are non-decreasing, and both $\Lambda_{<j}$ and $V(K)$ are decremental vertex sets by Theorem 3.1. At a high level, our algorithm can be described in one sentence:

Maintain a collection of ES-Trees of depth R^d_j rooted at every far core in $\bigcup_j \mathcal{F}_j$.

Below, we describe the data structure in more detail and analyze its correctness.

5.2.1 Maintaining Far Vertices and Far Cores In this subsection, we show an algorithm that maintains, for every vertex of G, whether it is a far vertex. It also maintains, for every core of $\bigcup_j \mathcal{F}_j$, whether it is a far core. Fix a layer $1 < j \leq r$. Let Z_j be a graph, whose vertex set is $V(G)$, and edge set contains all edges that have at least one endpoint in set $\Lambda_{\geq j}$. Equivalently, $E(Z_j)$ contains all edges incident to vertices with virtual degree at most h_j. We construct another graph Z'_j by adding a source vertex s_j to Z_j, and adding, for every vertex $v \in \Lambda_{<j}$, an edge (s, v) to this graph. We maintain an ES-Tree $\hat{T}_j$ in graph Z'_j, with root s_j, and distance bound $(R^d_j + 1)$. Observe that $v \in \Lambda_j$ is a far vertex iff $v \notin V(\hat{T}_j)$.

Notice that graph Z'_j, in addition to undergoing edge deletions, may also undergo edge insertions. Specifically, when a vertex x is moved from from $\Lambda_{<j}$ to $\Lambda_{\geq j}$ (that is, its virtual degree decreases from above h_j to at most h_j), then we may need to insert all edges that are incident to x into Z'_j. Note that edges connecting x to vertices in $\Lambda_{\geq j}$ already belong to Z'_j, so we only need to insert edges connecting x to vertices of $\Lambda_{<j}$. We insert all such edges Z'_j first, and only then delete the edge (s_j, x) from Z'_j. Observe that, for each such edge $e = (x, y) \in E(x, \Lambda_{<j})$, inserting e into Z'_j may not decrease the distance from s_j to x, or the distance from s_j to y, as both these distances are currently 1 and cannot be further decreased. It then follows that the insertion of the edge e does not decrease the distance of any vertex from s_j. Therefore, the edge insertions satisfy the conditions of the ES-Tree data structure.

As the total number of edges that ever appear in Z'_j is $O(nh_j\Delta)$ by Observation 3.3, the total update time for maintaining the data structure $\hat{T}_j$ is bounded by $O(nh_j\Delta R^d_j) = O(n^{2+1/k} D(\gamma(n))^{O(k)}) \leq \widehat{O}(n^{2+1/k} D)$ (we have used the fact that $h_j = \Delta^{r-j}$, $\Delta = n^{1/k}$, and $r \leq k + 1$).

The above data structure allows us to maintain, for every vertex of G, whether it is a far vertex. For every core $K \in \bigcup_j \mathcal{F}_j$, we simply maintain the number of vertices of K that are far vertices. This allows us to maintain, for every core $K \in \bigcup_j \mathcal{F}_j$, whether it is a far core. The time that is required for tracking

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

this information is clearly subsumed by the time for maintaining $\hat{T}_j$. Therefore, the total time that is needed to maintain the information about far vertices and far cores, over all layers j, is bounded by $\widehat{O}(n^{2+1/k}D)$.

5.2.2 Maintaining ES-Trees Rooted at Far Cores In this section, we define additional data structures that maintain ES-Trees that are rooted at the far cores, and analyze their total update time. Fix a layer $1 < j \leq r$. Let $K \in \mathcal{F}_j$ be a core in layer j, that is a far core. Let Z_j^K be the graph obtained from Z_j by adding a source vertex s_K, and adding, for every vertex $v \in V(K)$, an edge (s_K, v). Whenever a core K is created in layer j, we check if K is a far core. If this is the case, then we initialize an ES-Tree T_K in graph Z_j^K, with source s_K, and distance bound $(R_j^d + 1)$. We maintain this data structure until core K is destroyed. Additionally, whenever an existing core K becomes a far core for the first time, we initialize the data structure T_K, and maintain it until K is destroyed.

Observe that graph Z_j^K may undergo both edge insertions and deletions. As before, an edge may be inserted into Z_j^K only when some vertex x is moved from $\Lambda_{<j}$ to $\Lambda_{\geq j}$ (recall that vertices may only be removed from a core K after it is created). When vertex x moves from $\Lambda_{<j}$ to $\Lambda_{\geq j}$, we insert all edges connecting x to vertices of $\Lambda_{<j}$ into the graph Z_j^K. We claim that the insertion of such edges may not decrease the distance from s_K to any vertex $v \in V(T_K)$. In order to see this, observe that, since vertex x initially belonged to $\Lambda_{<j}$, and core K was a far core, $\mathsf{dist}_G(V(K), x) > R_j^d$. As edges are deleted from G and K, $\mathsf{dist}_G(V(K), x)$ may only grow. Therefore, when vertex x is moved to $\Lambda_{\geq j}$, its distance from the vertices of K remains greater than R_j^d, and so $\mathsf{dist}_{Z_j^K}(s_K, x) > R_j^d + 1$. As the depth of T_K is $R_j^d + 1$, inserting the edges of $E(x, \Lambda_{<j})$ does not affect the distances of the vertices that belong to the tree T_K from its root s_K.

Since, from by Observation 3.3, the total number of edges that may ever appear in Z_j^K is $O(nh_j\Delta)$, the total time required for maintaining the ES-Tree T_K is $O(nh_j\Delta) \cdot (R_j^d + 1)$. By Theorem 3.1, the total number of cores that are ever created in set $\mathcal{F}_j$ over the course of the entire algorithm the algorithm is at most $\widehat{O}(n\Delta/h_j)$. Therefore, the total update time that is needed in order to maintain trees T_K for cores $K \in \mathcal{F}_j$ is bounded by:

$$\begin{aligned} &O(nh_j\Delta R_j^d) \cdot \widehat{O}(n\Delta/h_j) \\ &= \widehat{O}(n^{2+2/k} D(\gamma(n))^{O(k)}) \\ &= \widehat{O}(n^{2+2/k}D). \end{aligned}$$

Summing this bound over all layers increases it by only factor $O(\log n)$.

5.2.3 Total update time We now bound the total update time of the algorithm. Recall that the total update time of the LCD data structure is bounded by $\widehat{O}(m^{1+1/q}\Delta^{2+1/q} \leq \widehat{O}(mn^{3/k})$, as $q = 10k$ and $\Delta = n^{1/k}$. Each of the remaining data structures takes total update time at most $\widehat{O}(n^{2+2/k}D)$. Therefore, the total update time of the algorithm is bounded by $\widehat{O}(n^{2+3/k}D)$.

5.2.4 Responding to Queries For any vertex $v \in \Lambda_{\geq j}$, we say that v is *covered* by an ES-Tree T_K iff $\mathsf{dist}_{Z_j}(V(K), v) \leq R_j^c$ (i.e. $\mathsf{dist}_{Z_j^K}(s_K, v) \leq R_j^c + 1$). For each $v \in \Lambda_{\geq j}$, we maintain a list of all ES-Trees T_K that cover it. Within the list of v, we maintain the core $K \in \mathcal{F}_{j_v}$ from the smallest layer index j_v such that T_K covers v. These indices can be explicitly maintained using the standard dictionary data structure such as balanced binary search trees. The time for maintaining such lists for all vertices is clearly subsumed by the time for maintaining the ES-Trees.

Responding to $\mathsf{path\text{-}query}_D(u, v)$. Given a pair of vertices u and v, let K_u be the core from smallest level j_u such that T_{K_u} covers u and let K_v be the core from smallest level j_v such that T_{K_v}covers v. Assume w.l.o.g. that $j_u \leq j_v$. If $v \notin T_{K_u}$, then we report that $\mathsf{dist}_G(u, v) > 2D$. Otherwise, compute the unique u-v path P in the tree T_{K_u}. This can be done in time in time $O(|P| \log n)$, as follows. We maintain two current vertices u', v', starting with $u' = u$ and $v' = v$. In every iteration, if the distance of u' from the root of T_{K_u} in tree T_{K_u} is less than the distance of v' from the root, we move v' to its parent in the tree; otherwise, we move u' to its parent. We continue this process, until we reach a vertex z that is a common ancestor of both u and v'. We denote the resulting u-v path by P. Notice that so far the running time of the algorithm is $O(|E(P)|)$. Next, we consider two cases. First, if z is not the root of the tree T_{K_u}, then P is a path in graph G, and we return P. Otherwise, the root of the tree s_{K_u} lies on path P. We let a and b be the vertices lying immediately before and immediately after s_{K_u} in P. We compute $Q = \mathsf{Short\text{-}Core\text{-}Path}(K_u, a, b)$ in time $(\gamma(n))^{O(q)}$. Finally, we modify the path P by replacing vertex s_{K_u} with the path Q, and merging the endpoints a, b of Q with the copies of these vertices on path P. The resulting path, that we denote by P', is a u-v path in graph G. We return this path as the response to the query. It is immediate to verify that the query time is $O(|E(P)| \log n) + (\gamma(n))^{O(q)} = \widehat{O}(|P|)$.

We now argue that the response of the algorithm to the query is correct.

Let P^* be the shortest path between u and v in graph G. Let x be a vertex of P^* that minimizes the

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

index j^* for which $x \in \Lambda_{j^*}$; therefore, $V(P^*) \subseteq \Lambda_{\geq j^*}$. We start with the following crucial observation.

Lemma 5.1 *There is a far core K' in some level $\Lambda_{j'}$, with $1 < j' \leq j^*$, such that $\mathsf{dist}_{Z_{j'}}(V(K'), x) \leq R^c_{j'} - D$.*

Proof. Let $x_1 = x$. We gradually construct a path connecting x_1 to a vertex in a far core K', as follows. First, using query $\mathsf{To\text{-}Core\text{-}Path}(x)$ of the LCD data structure, we can obtain a path of length at most α, connecting x_1 to a vertex a_1 lying in some core K_1, such that, if $K_1 \in \mathcal{F}_{j_1}$, then $j_1 \leq j^*$. If K_1 is a far core, then we are done. Otherwise, there is a vertex b_1 in K_1 which is not a far vertex. By using a query $\mathsf{Short\text{-}Core\text{-}Path}(K_1, a_1, b_1)$ of the LCD data structure, we obtain a path of length at most α connecting a_1 to b_1 inside the core K_1. As b_1 is not a far vertex, there must be some vertex $x_2 \in \Lambda_{<j_1}$, for which $\mathsf{dist}_{Z_{j_1}}(b_1, x_2) \leq R^d_{j_1}$. We repeat the argument for x_2 and subsequent vertices x_i, until we reach a vertex that lies in some far core K'. Note that, if $K' \in \mathcal{F}_{j'}$, then $j' > 1$ must hold, as $\Lambda_1 = \emptyset$. Observe that, for each i, the constructed paths that connect x_i and a_i, or connect a_i to b_i, or connect b_i to x_{i+1}, all lie inside $Z_{j'}$. By concatenating all these paths, we obtain a path in $Z_{j'}$, connecting x to a core of K'. The length of the path is bounded by:

$$\begin{aligned}
&(2\alpha + R^d_{j^*}) + (2\alpha + R^d_{j^*-1}) + \cdots + (2\alpha + R^d_{j'+1}) + \alpha \\
&\leq R^d_{j^*} + R^d_{j^*-1} + \cdots + R^d_{j'+1} + 2\alpha k \\
&= (3D + 2\alpha k)(1 + 2 + \cdots + 2^{r-(j'+1)}) + 2\alpha k \\
&= (3D + 2\alpha k)(2^{r-j'} - 1) + 2\alpha k \\
&= R^d_{j'} - 3D \\
&= R^c_{j'} - D
\end{aligned}$$

We conclude that that $\mathsf{dist}_{Z_{j'}}(V(K'), x) \leq R^c_{j'} - D$. ∎

We assume w.l.o.g. that x is closer to u than v, that is, $\mathsf{dist}_G(u, x) \leq \mathsf{dist}_G(v, x)$. Assume that P^* has length at most $2D$. As x lies on P^* and $V(P^*) \subseteq \Lambda_{\geq j^*}$, we get that $\mathsf{dist}_{Z_{j^*}}(u, x) \leq \frac{2D}{2} = D$. As Z_{j^*} is a subgraph of $Z_{j'}$, we conclude that $\mathsf{dist}_{Z_{j'}}(u, x) \leq \mathsf{dist}_{Z_{j^*}}(u, x) \leq D$. Using the triangle inequality together with Lemma 5.1, we get that $\mathsf{dist}_{Z_{j'}}(u, V(K')) \leq \mathsf{dist}_{Z_{j'}}(u, x) + \mathsf{dist}_{Z_{j'}}(x, V(K')) \leq R^c_{j'}$. In other words, tree $T_{K'}$ must cover u. Recall that we have let K_u be the core lying in smallest level j_u, such that T_{K_u} covers u. Therefore, $j_u \leq j'$ which implies that $V(P^*) \subseteq \Lambda_{\geq j_u}$. Therefore, path P^* is contained in Z_{j_u}. Moreover, as $R^d_{j_u} = R^c_{j_u} + 2D$ and $|P^*| \leq 2D$, vertex v must be contained in T_{K_u} as well. If this is not the case, then we can conclude that $|P^*| > 2D$. The same argument applies if the index j_v of the layer Λ_{j_v} to which the core K_v belongs is smaller than j_u.

Let P be the unique u-v path in the tree T_{K_u}. Clearly, $|P| \leq \mathsf{dist}_{T_{K_u}}(s_{K_u}, u) + \mathsf{dist}_{T_{K_u}}(s_{K_u}, v) \leq 2R^d_{j_u} \leq 2^k \cdot 3D + (\gamma(n))^{O(k)}$. If the root vertex s_{K_u} of the tree does not lie on the path P, then path P is a u-v path in graph G, whose length is bounded by $2^k \cdot 3D + (\gamma(n))^{O(k)}$; the algorithm then returns P. Otherwise, the algorithm replaces the vertex s_{K_u} with the path Q returned by the query $\mathsf{Short\text{-}Core\text{-}Path}(K_u, a, b)$ to the LCD data structure, where a and b are the vertices of P appearing immediately before and after s_{K_u} on it. As $|Q| \leq \alpha$, the length of returned path is bounded by $2R^d_{j_u} + \alpha \leq 2^k \cdot 3D + (\gamma(n))^{O(k)}$.

Responding to $\mathsf{dist\text{-}query}_D(u, v)$. The algorithm for responding to $\mathsf{dist\text{-}query}_D(u, v)$ is similar. As before, we let K_u be the core from smallest level j_u such that T_{K_u} covers u, and we let K_v be the core from smallest level j_v such that T_{K_v} covers v. Assume w.l.o.g. that $j_u \leq j_v$. If $v \notin T_{K_u}$, then we report that $\mathsf{dist}_G(u, v) > 2D$. Otherwise, we declare that $\mathsf{dist}(u, v) \leq 2^k \cdot 3D + (\gamma(n))^{O(k)}$. The correctness of this algorithm follows immediately from the analysis of the algorithm for responding to $\mathsf{path\text{-}query}_D(u, v)$. The algorithm can be implemented to run in time $O(1)$ if we store, together with every vertex $v \in V(G)$, the list of the cores that cover v, sorted by the index j of the set $\mathcal{F}_j$ to which the core belongs. It is easy to see that time that is required to maintain this data structure is subsumed by the total update time of the algorithm that was analyzed previously.

References

[ACT14] Ittai Abraham, Shiri Chechik, and Kunal Talwar. Fully dynamic all-pairs shortest paths: Breaking the O (n) barrier. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 28. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014. 3

[AMV20] Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. Circulation control for faster minimum cost flow in unit-capacity graphs. *arXiv preprint arXiv:2003.04863*, 2020. 3

[BC16] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the O(mn) bound. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 389–397. ACM, 2016. 2, 5

[BC17] Aaron Bernstein and Shiri Chechik. Deterministic partially dynamic single source shortest paths for sparse graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 453–469. SIAM, 2017. 2

[Ber16] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM Journal on Computing*, 45(2):548–574, 2016. 2, 3

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

[Ber17] Aaron Bernstein. Deterministic partially dynamic single source shortest paths in weighted graphs. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 80. Schloss Dagstuhl-Leibniz-Center for Computer Science, 2017. 2, 3, 10, 11

[BGS20] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. 2020. To appear at FOCS'20. 4, 5, 9

[BHI15] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 785–804, 2015. 2

[BHN16] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 398–411, 2016. 2

[BHS07] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *J. Algorithms*, 62(2):74–92, 2007. 3

[BK19] Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2+\epsilon)$-approximate minimum vertex cover in $o(1/\epsilon^2)$ amortized update time. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1872–1885, 2019. 2

[BR11] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 1355–1365, 2011. 2, 3, 4, 10

[BvdBG+20] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. *CoRR*, abs/2004.08432, 2020. 2, 3, 4

[CGL+19] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. *CoRR*, abs/1910.08025, 2019. 2, 9

[Che18] Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *Proc. of the IEEE 59th Annual Symposium on Foundations of Computer Science*, 2018. 3, 4, 10

[CK19] Julia Chuzhoy and Sanjeev Khanna. A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems. In *STOC 2019, to appear*, 2019. 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

[CQ17] Chandra Chekuri and Kent Quanrud. Approximating the held-karp bound for metric TSP in nearly-linear time. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 789–800, 2017. 2

[DHZ00] Dorit Dor, Shay Halperin, and Uri Zwick. All-pairs almost shortest paths. *SIAM J. Comput.*, 29(5):1740–1759, 2000. 3

[Din06] Yefim Dinitz. Dinitz' algorithm: The original version and Even's version. In *Theoretical computer science*, pages 218–240. Springer, 2006. 5

[ES81] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981. 2, 3, 5

[FHN14a] Sebastian Forster, Monika Henzinger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 146–155, 2014. 2, 3, 4, 10

[FHN14b] Sebastian Forster, Monika Henzinger, and Danupon Nanongkai. A subquadratic-time algorithm for decremental single-source shortest paths. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1053–1072, 2014. 2

[FHN16] Sebastian Forster, Monika Henzinger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the O(mn) barrier and derandomization. *SIAM Journal on Computing*, 45(3):947–1006, 2016. Announced at FOCS'13. 2, 3, 4, 10, 13

[FHNS15] Sebastian Forster, Monika Henzinger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30, 2015. 2, 3

[GWN20] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Deterministic algorithms for decremental approximate shortest paths: Faster and simpler. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2522–2541. SIAM, 2020. 2, 4, 10, 13

[HdLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001. 4, 5

[HK95] Monika Rauch Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 664–672. IEEE,

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

1995. 5

[LS14] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in õ(vrank) iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 424–433, 2014. 3

[Mad10] Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 121–130, 2010. 2

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and O($n^{1/2 - \epsilon}$)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129, 2017. 2, 5

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 950–961, 2017. 2, 5

[RZ11] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011. 3

[RZ12] Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM Journal on Computing*, 41(3):670–683, 2012. 3

[San05] Piotr Sankowski. Subquadratic algorithm for dynamic shortest distances. In *International Computing and Combinatorics Conference*, pages 461–470. Springer, 2005. 2

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. 2

[SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2616–2635, 2019. 5, 9

[TZ01] M. Thorup and U. Zwick. Approximate distance oracles. *Annual ACM Symposium on Theory of Computing*, 2001. 3

[vdBNS19] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 456–480, 2019. 2

[Waj20] David Wajc. Rounding dynamic matchings against an adaptive adversary. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 194–207, 2020. 2

[WN17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1130–1143. ACM, 2017. Full version at arXiv:1611.02864. 2

[Zwi98] Uri Zwick. All pairs shortest paths in weighted directed graphs-exact and almost exact algorithms. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*, pages 310–319. IEEE, 1998. 2

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited