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.

*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.

Contents

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 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

[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 $O(n^{2-\epsilon})$ query time, for any constant $\epsilon > 0$, the total update time of $\Omega(n^{3-o(1)})$ is required in graphs with $m = \Theta(n^2)$.

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 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:*

[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 separation holds for decremental exact APSP.

- $\mathsf{dist\text{-}query}(s,v)$: *in* $O(\log\log(nL))$ *time return an estimate* $\widetilde{\mathsf{dist}}(u,v)$*, with* $\mathsf{dist}_G(s,v) \leq \widetilde{\mathsf{dist}}(s,v) \leq (1+\epsilon)\mathsf{dist}_G(s,v)$*; 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 Appendices C and D 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 $k \geq 1$, 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 $(1+\epsilon)$-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**

[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 $m=\Omega(n^2)$.

[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.

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.

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* (from Section 3.2), 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

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

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 expander" 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. We exploit this data structure to obtain our algorithms for SSSP and APSP in Section 4 and Section 5, respectively. The new cut/flow applications of our SSSP algorithm (that exploit known reductions) appear in Appendices C and D.

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 $\Phi(G) \geq \varphi$.

Suppose we are given a graph G and a sub-graph $G' \subseteq G$. We say that G' is a *strong φ-expander with respect to G* iff for every partition $(S, \overline{S})$ of $V(G')$ into non-empty subsets, $\frac{\delta_{G'}(S)}{\min\{\mathrm{vol}_G(S), \mathrm{vol}_G(\overline{S})\}} \geq \varphi$ (note that in the denominator, the volumes of the sets $S, \overline{S}$ of vertices are taken in graph G, not in G' as in the definition of φ-expansion of G'). It is easy to verify that, if G' is a strong φ-expander with respect to G, then it is also a φ-expander. The following two simple observations follow from the definition of a strong φ-expander.

Observation 2.1 *Let G be a graph such that for all $v \in V(G)$, $\deg_G(v) \geq h$ for some $h > 0$, and let $G' \subseteq G$ be a strong φ-expander with respect to G, for some $0 < \varphi < 1$, such that $|V(G')| \geq 2$. Then, for every vertex $v \in V(G')$, $\deg_{G'}(v) \geq \varphi h$.*

Proof: Assume otherwise, and let $v \in V(G')$ be any vertex with $\deg_{G'}(v) < \varphi h$. Consider the cut $(S, \overline{S})$ of $V(G')$, where $S = \{v\}$, and $\overline{S} = V(G') \setminus \{v\}$. Then $\mathrm{vol}_G(S) \geq h$, $\mathrm{vol}_G(\overline{S}) \geq h$, but $\delta(S) = \deg_{G'}(v) < \varphi h$, contradicting the fact that G' is a strong φ-expander with respect to G. □

Observation 2.2 *Let G be a graph and let G' be a sub-graph of G containing at least two vertices, such that G' is a strong φ-expander with respect to G, for some $0 < \varphi < 1$. Then, for every vertex $v \in V(G')$ with $\deg_G(v) \leq \mathrm{vol}_G(V(G'))/2$, $\deg_{G'}(v) \geq \varphi \deg_G(v)$ must hold.*

Proof: Consider the cut $(\{v\}, V(G') \setminus \{v\})$ in G'. Then $\frac{\deg_{G'}(v)}{\deg_G(v)} = \frac{\delta_{G'}(\{v\})}{\min\{\mathrm{vol}_G(\{v\}), \mathrm{vol}_G(V(G') \setminus \{v\})\}} \geq \varphi$ must hold, as G' is a strong φ-expander with respect to G. Therefore, $\deg_{G'}(v) \geq \varphi \deg_G(v)$. □

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 ($\mathsf{ES\text{-}Tree}$) 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, $\mathsf{ES\text{-}Tree}$ 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 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 $\mathsf{Proc\text{-}Degree\text{-}Pruning}(H, d)$ and denote by A the output of the procedure. The following observation was implicitly shown in [CK19]; for completeness, we provide its proof in Appendix.

Observation 2.3 *Let A be the outcome of procedure $\mathsf{Proc\text{-}Degree\text{-}Pruning}(H, d)$, 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 2.4 *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 2.3, 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 2.3, 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.*

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 Appendix.

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 Section B.2 of the Appendix, 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 component 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.

- $\mathsf{To\text{-}Core\text{-}Path}(u)$: 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 $u_{i+1} \in \Lambda_{\leq j}$.

- $\mathsf{Short\text{-}Core\text{-}Path}(K, u, v)$: 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 3.4 (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 $\mathsf{Short\text{-}Path}(j, u, v)$ in time $O(\log n)$ if u and v 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 $\mathsf{To\text{-}Core\text{-}Path}(u)$ with query time $O(|P|)$, where P is the returned path, and $\mathsf{Short\text{-}Core\text{-}Path}(K, u, v)$ 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 3.4 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 3.4. 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 $\mathsf{Short\text{-}Path}(j, u, v)$ 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 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 size is 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 $\widetilde{O}(|V(K)|)+\widehat{O}(1)$, 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-distances regime exploits the LCD data structure in a similar way as in 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-distances 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 remainder of this section is dedicated to the proof of Theorem 3.4. 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.

Implementation of Layered Core Decomposition

In the remainder of this section, we provide the proof of Theorem 3.4 by showing an implementation of the LCD data structure, which is the central technical tool of this paper. We start by observing that all layers $\Lambda_1, \ldots, \Lambda_r$ can be maintained in near linear time:

Observation 3.5 *There is a deterministic algorithm, that, given an n-vertex m-edge graph G undergoing edge deletions and parameter $\Delta \geq 2$, maintains the partition $(\Lambda_1, \ldots, \Lambda_r)$ of $V(G)$ into layers. Additionally, for every vertex $v \in V$ and index $1 \leq j \leq r+1$, the algorithm maintains a list of all neighbors of v that belong to Λ_j. The total update time of the algorithm is $\tilde{O}(m+n)$.*

Proof: We maintain the partition $(\Lambda_1, \ldots, \Lambda_{r+1})$ of $V(G)$ into layers, as graph G undergoes edge deletions, as follows. We run Alg-Maintain-Pruned-Set(G, h_j) for maintaining the vertex set A_j, for all $1 \leq j \leq r$ in parallel. Whenever a vertex $v \in \Lambda_j$ is deleted from A_j by this algorithm, we update its layer accordingly. It is easy to verify that the total update time for maintaining the partition of $V(G)$ into layers is $O((|E(G)| + |V(G)|) \cdot r) = \tilde{O}(m+n)$.

Additionally, for every vertex $v \in V(G)$ and index $1 \leq j \leq r+1$, we maintain a list of all neighbors of v that lie in Λ_j. In order to maintain this list, whenever a vertex $u \in \Lambda_i$ is removed from set A_{i-1}, we inspect the lists of all neighbors of u, and for each such neighbor v, we move u to the list of neighbors of v corresponding to the new layer of u. Therefore, whenever a virtual degree of a vertex u decreases, we spend $O(\deg_G(u))$ time to update the lists of its neighbors. As virtual degrees can decrease at most r times for every vertex, the total update time for initializing and maintaining these lists is $O(|E(G)| + |V(G)|) \cdot r = \tilde{O}(m+n)$. □

The remainder of the section is organized as follows. First, we list some known tools related to expanders in Section 3.1 and then, in Section 3.2, provide a new tool, called a short-path oracle on decremental expanders that will be useful for Short-Core-Path queries. One of our key ideas is to further partition each layer Λ_j into *sublayers* $\Lambda_{j,1}, \ldots, \Lambda_{j,L_j}$. We describe the structure of the sublayers and the invariants that we maintain for each sublayer in Section 3.3. For every sublayer $\Lambda_{j,\ell}$, the execution of the algorithm is partitioned into phases with respect to that sublayer, that we refer to as (j,ℓ)-*phases*. At the beginning of each (j,ℓ)-phase, we compute a core decomposition of graph $G[\Lambda_{j,\ell}]$ and obtain a collection $\mathcal{F}_{j,\ell}$ of cores for the sublayer $\Lambda_{j,\ell}$. Section 3.4 describes the algorithm for computing the core decompositions. The description of an algorithm that we use to maintain each core and to support Short-Core-Path queries on each core is shown in Section 3.5. During each (j,ℓ)-phase, vertices can move between the sublayers of layer j in a non-monotone manner (in contract to the fact that every vertex can only move from higher to lower layers). We describe how vertices are moved between sublayers in Section 3.6 and state the key technical lemma that bounds the total number of times vertices may move between sublayers. We then bound the total number of cores ever created by the algorithm in Section 3.7; this bound is crucial for our LCD data structure. In Section 3.9, we show an algorithm for processing To-Core-Path queries. We provide additional technical details for maintaining all necessary data structures in Section 3.10 and Section 3.11. Finally, we describe the algorithm for responding to Short-Path queries in Section 3.12.

3.1 Known Expander-Related Tools

In this subsection we describe several expander-related tools, that mostly follow from previous work, that our algorithm uses.

Expander Decomposition. The following theorem can be obtained immediately from the recent deterministic algorithm of [CGL+19] for computing a (standard) expander decomposition in almost-linear time; the proof is deferred to Appendix B.3. As before, we denote $\gamma(n) = \exp(O(\log^{3/4} n)) = n^{o(1)}$.

Theorem 3.6 *There is a deterministic algorithm, that, given a connected graph $G = (V, E)$ with n vertices and m edges, and a parameter $0 \leq \varphi \leq 1$, computes a partition of V into disjoint subsets $V_1, \ldots, V_k$, such that $\sum_{i=1}^{k} \delta(V_i) \leq \gamma(m) \cdot \varphi m$, and, for all $1 \leq i \leq k$, $G[V_i]$ is a strong φ-expander with respect to G. The running time of the algorithm is $\widehat{O}(m)$.*

Expander Pruning. In the following theorem, we consider the setting where we are given as input a graph $G = (V, E)$, with $|E| = m$. Intuitively, we hope that G is a φ-expander for some $0 \leq \varphi \leq 1$, though it may not be the case. We assume that G is represented by its adjacency list. We also assume that we are given an input sequence $\sigma = (e_1, e_2, \ldots, e_k)$ of online edge deletions, and we denote by G_i the graph G at time i, that is, G_0 is the original graph G, and for all $1 \leq i \leq k$, $G_i = G \setminus \{e_1, \ldots, e_i\}$. Our goal is to maintain a set $S \subseteq V$ of vertices, such that, intuitively, if we denote by S_i the set S at time i (that is, after the deletion of the first i edges from G), then $G[V \setminus S_i]$ is large enough compared to G. Moreover, if G was a φ-expander, then for all i, $G[V \setminus S_i]$ remains a strong enough expander. We also require that the set S is incremental, that is, $S_{i-1} \subseteq S_i$ for all i. The following theorem, proved in [SW19] allows us to do so.

Theorem 3.7 (Restatement of Theorem 1.3 in [SW19]) *There is a deterministic algorithm, that, given an access to the adjacency list of a graph $G = (V, E)$ with $|E| = m$, a parameter $0 < \varphi \leq 1$, and a sequence $\sigma = (e_1, e_2, \ldots, e_k)$ of $k \leq \varphi m/10$ online edge deletions, maintains a vertex set $S \subseteq V$ with the following properties. Let G_i be the graph G after the edges $e_1, \ldots, e_i$ have been deleted from it; let $S_0 = \emptyset$ be the set S at the beginning of the algorithm, and for all $0 < i \leq k$, let S_i be the set S after the deletion of $e_1, \ldots, e_i$. Then, for all $1 \leq i \leq k$:*

- *$S_{i-1} \subseteq S_i$;*
- *$\mathrm{vol}_G(S_i) \leq 8i/\varphi$;*
- *$|E(S_i, V \setminus S_i)| \leq 4i$; and*
- *if G is a φ-expander, then $G_i[V \setminus S_i]$ is a strong $\varphi/6$-expander with respect to G.*

The total running time of the algorithm is $O(k \log m/\varphi^2)$.

Embeddings. Let G, W be two graphs with $V(W) \subseteq V(G)$. A set $\mathcal{P}$ of paths in G is called an *embedding of W into G* if, for every edge $e = (u, v) \in E(W)$, there is a path $\mathrm{path}(u, v) \in \mathcal{P}$, such that $\mathrm{path}(u, v)$ is a u-v path in G. We say that the *length* of the embedding $\mathcal{P}$ is l if the length of every path in $\mathcal{P}$ is at most l, and we say that the *congestion* of the embedding is η iff every edge of G participates in at most η paths in $\mathcal{P}$. If embedding $\mathcal{P}$ has length l and congestion η, then we sometimes call it an (l, η)-embedding, and we say that W (l, η)-embeds into G.

The following algorithm allows us to quickly embed a smaller expander into a given expander; the proof appears in Appendix B.4. Recall that we denoted $\gamma(n) = \exp(O(\log^{3/4} n))$.

Theorem 3.8 *There is a deterministic algorithm that, given an n-vertex m-edge graph G which is a φ-expander, and a terminal set $T \subseteq V(G)$, computes a graph W with $V(W) = T$ and maximum vertex degree $O(\log |T|)$ such that W is a $(1/\gamma(|T|))$-expander. The algorithm also computes a (l, η)-embedding $\mathcal{P}$ of W into G with $l = O(\varphi^{-1} \log m)$ and $\eta = O(\varphi^{-2} \log^2 m)$. The running time of the algorithm is $\tilde{O}(m \cdot \gamma(|T|)/\varphi^3)$.*

3.2 A New Tool: Short-Path Oracle for Decremental Expanders

Based on known expander-related tools from the previous section, we provide a new tool, that we call a *short-path oracle on decremental expanders*. This will be a key tool for Short-Core-Path queries. We believe that the techniques used in this section are of independent interest as they are quite generic. In fact, they have already been subsequently generalized to directed graphs in [BGS20]. We fix the following parameters that will be used throughout this section. We set the parameters as follows:

$$\gamma = \gamma(m) = \exp(O(\log^{3/4} m)) = n^{o(1)}; \tag{1}$$

and

$$\varphi = 1/(c \cdot \gamma), \tag{2}$$

where c is a large enough constant.

Below, we say that a vertex set S is *incremental* if vertices in S can never leave S as time progresses.

Theorem 3.9 *There is a deterministic algorithm that, given an m-edge n-vertex φ-expander G undergoing a sequence at most $\varphi|E(G)|/10$ edge deletions, and a parameter $q > 1$, maintains an incremental* vertex set $S \subseteq V(G)$, *such that, if we denote by $G^{(0)}$ the graph G before any edge deletions, then, for every $t > 0$, after t edges are deleted from G, $\mathrm{vol}_{G^{(0)}}(S) \leq O(t/\varphi)$ holds and $G \setminus S$ is a strong $\varphi/6$-expander with respect to $G^{(0)}$. The algorithm also supports the following query: given a pair of vertices $u, v \in V(G) \setminus S$, return a simple u-v path P in $G \setminus S$ of length at most $(\gamma(m))^{O(q)}$, with query time $(\gamma(m))^{O(q)}$. The total update time of the algorithm is $O(m^{1+1/q}(\gamma(n))^{O(q)})$.*

The remainder of this section is dedicated to proving Theorem 3.9.

Throughout the algorithm, m denotes the number of edges in the original φ-expander graph G, and the parameter $\varphi = 1/(c\gamma(m))$ remains unchanged. As our main tools, we employ the Expander Pruning Algorithm from Theorem 3.7, and the algorithm from Theorem 3.8 that allows us to embed a smaller expander into a given expander. We use parameters $l = O(\log m/\varphi)$ and $\eta = O(\log^2 m/\varphi^2)$. The idea of the algorithm is to maintain a hierarchy of expander graphs $G_1, \ldots, G_q$, where for all $1 \leq i < q$, graph G_i contains $\lceil m^{i/q} \rceil$ vertices, and it is a $\varphi/6$-expander; we set $G_q = G$. We will also maintain an (l, η)-embedding $\mathcal{P}_i$ of each such graph G_i into graph G_{i+1}. Initially, for all $1 \leq i < q$, both the graph G_i and its embedding into G_{i+1} are computed using Theorem 3.8. Additionally, we maintain an ES-Tree in graph G_{i+1}, rooted at the vertex set $V(G_i)$, with distance threshold $O(\log n/\varphi)$. For every edge $e \in E(G_i)$, we will maintain a list $J_i(e)$ of all edges $e' \in E(G_{i-1})$, such that the embedding of e' in $\mathcal{P}_{i-1}$ contains the edge e; recall that $|J(e)| \leq \eta$ must hold. Whenever edge e is deleted from graph G_i, this will trigger the deletion of all edges in its list $J_i(e)$ from graph G_{i-1}. Lastly, we use the

Algorithm 1 `InitializeExpander`(i)

Assertion: G_i is a $\varphi/6$-expander.

1. If $i = 1$, then initialize an ES-Tree T_1 in G_1, rooted at an arbitrary vertex, with distance threshold $O(\varphi^{-1} \log n)$; return.
2. If $i = q$, then let X_q be an arbitrary subset of the set $\{x_e \mid e \in E(G)\}$ of vertices of G'_q of cardinality $\lceil m^{(q-1)/q} \rceil$; otherwise, set $G'_i = G_i$, and let X_i be any subset of $V(G'_i)$ of cardinality $\lceil m^{(i-1)/q} \rceil$.
3. Using the algorithm from Theorem 3.8, compute an expander G_{i-1} over vertex set X_i, and its (l, η)-embedding $\mathcal{P}_{i-1}$ into G'_i.
4. For every edge $e \in E(G_i)$, initialize a list $J_i(e)$ of all edges of G_{i-1} whose embedding path in $\mathcal{P}_{i-1}$ contains e.
5. Initialize the expander pruning algorithm from Theorem 3.7 on G_{i-1}, that will maintain a pruned vertex set $S_{i-1} \subseteq V(G_{i-1})$.
6. Initialize an ES-tree T_i in G'_i rooted at X_i, with distance threshold $O(\varphi^{-1} \log n)$.
7. Call `InitializeExpander`$(i-1)$.

algorithm from Theorem 3.7 in order to maintain, for every expander G_i, the set S_i of "pruned-out" vertices. When set S_i becomes too large, we re-initialize the graphs $G_i, G_{i-1}, \ldots, G_1$.

The outcome of the algorithm is vertex set $S = S_q$, the pruned-out set that we maintain for the expander $G_q = G$. Recall that $G^{(0)}$ denotes the graph G before any edge deletions. Theorem 3.7 directly guarantees that $G \setminus S$ is a strong $\varphi/6$-expander with respect to $G^{(0)}$ and $\mathrm{vol}_{G^{(0)}}(S) \leq O(t/\varphi)$ after t edge deletions as desired.

We note that, since G_q may be a high-degree graph, it is convenient to define a new graph G'_q, that is obtained from G_q by sub-dividing every edge e of $G_q = G$ with a new vertex v_e. We let $X = \{v_e \mid e \in E(G)\}$. It is easy to verify that G'_q remains a $\varphi/2$-expander.

In Algorithm 1 we describe the implementation of the algorithm `InitializeExpander`(i); the algorithm initializes the data structures for expander G_{i-1}, assuming that expander G_i is already defined. The algorithm then recursively calls to `InitializeExpander`$(i-1)$. At the beginning of the algorithm, we initialize the whole data structure by calling `InitializeExpander`(q). If, over the course of the algorithm, for some $1 \leq i < q$, the number of edges deleted from G_i exceeds $\varphi|E(G_i)|/10$, we will call `InitializeExpander`$(i-1)$.

We denote by $G^{(0)}_{i-1}$ the expander graph created by Procedure `InitializeExpander`(i). For all $d > 0$, we denote by $G^{(d)}_{i-1}$ the graph that is obtained from $G^{(0)}_{i-1}$ after d edge deletions from G. As d increases, our algorithm maintains the graph $G_{i-1} = G^{(d)}_{i-1} \setminus S_{i-1}$. By Theorem 3.7, as long as $d \leq \varphi|E(G_i)|/10$, graph G_{i-1} remains a $(\varphi/6)$-expander.

When some edge e is deleted from graph G, we call Algorithm `Delete`(q, e). The algorithm may recursively call to procedure `Delete`(i, e') for other expander graphs G_i and edges e'. The algorithm `Delete`(i, e') is shown in Algorithm 2.

We bound the total update time of the algorithm in the following lemma.

Algorithm 2 `Delete`(i, e) where $e \in E(G_i)$

1. If $i = 1$, delete e from graph G_1. Recompute an ES-Tree T_1 in graph G_1, up to depth $O(\log n/\varphi)$, rooted at any vertex; return.
2. Delete e from G_i. Update the pruned-out vertex set S_i using Theorem 3.7.
3. Let D_i^{new} denote the set of edges that were just removed from G_i. That is, D_i^{new} contains e and all edges incident to vertices that were newly added into S_i.
4. For each $e \in D_i^{new}$, for every edge $e' \in J_i(e)$, call `Delete`$(i-1, e')$;
5. If the total number of edge deletions from $G_i^{(0)}$ exceeds $\varphi|E(G_i^{(0)})|/10$, call `InitializeExpander`$(i+1)$.

Lemma 3.10 *The total update time of the algorithm is* $O(m^{1+1/q}(\gamma(n))^{O(q)})$.

Proof: Fix an index $1 \leq i \leq q$. We partition the execution of the algorithm into *level-i stages*, where each level-i stage starts when `InitializeExpander`$(i+1)$ is called, and terminates just before the subsequent call to `InitializeExpander`$(i+1)$. Recall that, over the course of a level-i stage, at most $\varphi|E(G_i^{(0)})|/10$ edges are deleted from the graph $G_i^{(0)}$. We now bound the running time that is needed in order to initialize and maintain the level-i data structure over the course of a single level-i stage. This includes the following:

- Computing expander G_i and its (l, η)-embedding $\mathcal{P}_{i-1}$ into G'_{i+1} using the algorithm from Theorem 3.8; the running time is $\tilde{O}(|E(G_{i+1})| \cdot \gamma(m)/\varphi^3) = O\left(m^{(i+1)/q} \cdot (\gamma(n))^{O(1)}\right)$.
- Initializing the lists $J_{i+1}(e)$ for edges $e \in G_{i+1}$: the time to initialize all such lists is bounded by the time needed to compute the embedding $\mathcal{P}_i$, which is in turn bounded by $O\left(m^{(i+1)/q} \cdot (\gamma(n))^{O(1)}\right)$.
- Initializing and maintaining the ES-Tree T_{i+1}: the running time is $O(|E(G_{i+1})| \cdot \log n/\varphi) \leq O\left(m^{(i+1)/q} \cdot (\gamma(n))^{O(1)}\right)$.
- Running the algorithm for expander pruning on the expander G_i; from Theorem 3.7, the running time is $\tilde{O}(|E(G_i^{(0)})/\varphi) \leq O\left(m^{i/q} \cdot (\gamma(n))^{O(1)}\right)$, since the number of edge deletions is bounded by $\varphi|E(G_i^{(0)})|/10$.
- The remaining work, executed by `Delete`(i, e), for every edge e that is deleted from graph G_i (including edges incident to the vertices of the pruned out set S_i), requires $O(\eta)$ time per edge, with total time $O(|E(G_i^{(0)})| \cdot \eta) \leq O\left(m^{i/q} \cdot (\gamma(n))^{O(1)}\right)$.

Therefore, the total time that is needed in order to initialize and maintain the level-i data structure over the course of a single level-i stage is $O\left(m^{(i+1)/q} \cdot (\gamma(n))^{O(1)}\right)$ Note that we did not include in this running time the time required for maintaining level-$(i-1)$ data structures, that is, calls to `InitializeExpander`(i) and `Delete`$(i-1, e)$.

Next, we bound the total number of level-i stages. Consider some index $1 < i' \leq q$, and consider a single level-i' stage. Recall that, over the course of this stage, at most $d_{i'} = \varphi|E(G_{i'}^{(0)})|/10$ edge deletions from graph $G_{i'}^{(0)}$ may happen. From Theorem 3.7, over the course of the level-i' stage, the total volume of edges that are incident to the pruned-out vertices in S_i is bounded $O(d_i/\varphi)$. As

Algorithm 3 $\texttt{Query}(i, u, v)$ where $u, v \in V(G_i)$

1. If $i = 1$, return the unique u-v path in T_1.
2. Compute, in T_i, a unique path $Q_{uu'}$ connecting u to some vertex $u' \in X_i$, and a unique path $Q_{v'v}$ connecting v to some vertex $v' \in X_i$ to v.
3. If $v' = u'$, set $R_{u'v'} = \emptyset$; otherwise set $R_{u'v'} = \texttt{Query}(i-1, u', v')$.
4. Let $Q_{u'v'}$ be a path in G_i obtained by concatenating, for all edges $e' \in R_{u'v'}$, the corresponding path $P(e') \in \mathcal{P}_i$ from the embedding of G_{i-1} into G'_i.
5. Return $Q_{uv} = Q_{uu'} \circ Q_{u'v'} \circ Q_{v'v}$. (Note that for $i = q$, $Q_{u,v}$ is a path in graph G'_q, that was obtained from G_q by subdividing its edges; it is immediate to turn it into a corresponding path in G_q.)

the embedding $\mathcal{P}_{i'}$ of $G_{i'-1}$ into G'_i has congestion at most η, this can cause at most $O(\eta d_i/\varphi)$ edge deletions from graph $G^{(0)}_{i'-1}$. As a single level-$(i'-1)$ stage requires $\varphi|E(G^{(0)}_{i'-1})|/10$ edge deletions from $G^{(0)}_{i'-1}$, the number of level-$(i'-1)$ stages that are contained in a single level-i' stage is bounded by:

$$\frac{O(d_{i'} \cdot \eta/\varphi)}{\varphi|E(G^{(0)}_{i'-1})|/10} \leq \frac{O(|E(G^{(0)}_{i'})| \cdot \log^3 m)}{\varphi^3 \cdot |E(G^{(0)}_{i'-1})|} \leq O(m^{1/q} \cdot (\gamma(n))^{O(1)}).$$

Since we only need to support at most $\varphi|E(G)|/10$ edge deletions from the original graph G, there is only a single level-q stage. Therefore, for every $1 \leq i < q$, the total number of level-i stages is bounded by: $O(m^{(q-i)/q} \cdot (\gamma(n))^{O(q-i)})$. We conclude that the total amount of time required for maintaining level-i data structure is bounded by:

$$O\left(m^{(i+1)/q} \cdot (\gamma(n))^{O(1)}\right) \cdot O\left(m^{(q-i)/q} \cdot (\gamma(n))^{O(q-i)}\right) \leq O\left(m^{1+1/q} \cdot (\gamma(n))^{O(q-i)}\right).$$

Summing this up over all $1 \leq i \leq q$, we get that the total update time of the algorithm is $O\left(m^{1+1/q} \cdot (\gamma(n))^{O(q)}\right)$, as required. □

Next, we provide an algorithm for responding to the short-path query between a pair u, v of vertices. The algorithm calls $\texttt{Query}(q, u, v)$, that is described in Algorithm 3, which recursively calls $\texttt{Query}(i, u', v')$ for $i < q$. The idea of the algorithm is simple: we use the ES-Tree T_q in graph G_q in order to compute two paths: one path connecting u to some vertex $u' \in X_q$, and one path connecting v to some vertex $v' \in X_q$, and then recursively call the short-path query for the pair u', v' of vertices in the expander G_{q-1}; we then use the embedding $\mathcal{P}_q$ of G_{q-1} into G_q to convert the resulting path into a u'-v' path in G_q. The final path connecting u to v is obtained by concatenating the resulting three paths.

The following lemma summarizes the guarantees of the algorithm for processing short-path queries.

Lemma 3.11 *Given any pair of vertices $u, v \in V(G) \setminus S$, algorithm $\texttt{Query}(q, u, v)$ returns a (possibly non-simple) u-v path Q in $G \setminus S$, of length $(\gamma(n))^{O(q)}$, in time $O(|Q|)$.*

Proof: Let $\text{Len}(i)$ be the maximum length of the path in G_i returned by $\texttt{Query}(i, u, v)$. As G_i is always a $\varphi/6$-expander by Theorem 3.7, it is immediate to verify that the diameter of G_i is $O(\varphi^{-1} \log n)$, and

so the ES-Tree tree T_i spans graph G_i. Consider Algorithm 3. Let $Q_{uu'}$ and $Q_{v'v}$ be the path in G'_i from u to $u' \in X_{i-1}$ and the path in G'_i from $v' \in X_{i-1}$ to v. As T_i spans G'_i, $Q_{uu'}$ and $Q_{v'v}$ do exist. Let $R_{u'v'} = \texttt{Query}(i-1, u', v')$ where $|R_{u'v'}| \leq \mathrm{Len}(i-1)$. Let $Q_{u'v'}$ be obtained by concatenating $\mathrm{path}(e')$ for each $e' \in R_{u'v'}$ where $\mathrm{path}(e') \in \mathcal{P}_i$ is the corresponding embedding path of e'. We have $|Q_{u'v'}| \leq \ell \cdot |R_{u'v'}|$. It is clear that the concatenation $Q_{uu'} \circ Q_{u'v'} \circ Q_{v'v}$ is indeed a u-v path in G'_i and hence in G_i. The length of this path is at most

$$\mathrm{Len}(i) = O(\varphi^{-1} \log n) + O(\varphi^{-1} \log n) \cdot \mathrm{Len}(i-1).$$

Solving the recursion gives us $\mathrm{Len}(i) = (\gamma(n))^{O(i)}$. So $\texttt{Query}(q, u, v)$ returns a u-v path of length $(\gamma(n))^{O(q)}$. Observe that the query time is proportional to the number of edges on the returned path. □

Lastly, observe that a path Q connecting the given pair u, v of vertices, that is returned by algorithm $\texttt{Query}(q, u, v)$ may not be simple. It is easy to turn Q into a simple path Q', in time $O(|Q|)$, by removing all cycles from Q. The final path Q' is guaranteed to be simple, of length $(\gamma(n))^{O(q)}$, and the query time is bounded by $(\gamma(n))^{O(q)}$, as required.

3.3 Sublayers and Phases

In this subsection we focus on a single layer Λ_j, for some $1 < j \leq r$. Recall that we have denoted by $n_{\leq j}$ the number of vertices that belonged to set $\Lambda_{\leq j}$ at the beginning of the algorithm, before any edges were deleted from the input graph. We let L_j be the smallest integer ℓ, such that $n_{\leq j}/2^{\ell-1} \leq h_j/2$; observe that $L_j \leq \log n$. We further partition vertex set Λ_j into subsets $\Lambda_{j,1}, \Lambda_{j,2}, \ldots, \Lambda_{j,L_j}$. We refer to each resulting vertex set $\Lambda_{j,\ell}$ as *sublayer* (j, ℓ). For indices $1 \leq \ell \leq \ell' \leq L_j$, we say that sublayer $\Lambda_{j,\ell}$ is *above* sublayer $\Lambda_{j,\ell'}$. The last sublayer Λ_{j,L_j}, that we also denote for convenience by Λ_j^-, is called the *buffer* sublayer. For convenience, we also use the shorthand $\Lambda_{j,\leq \ell} = \bigcup_{\ell'=1}^{\ell} \Lambda_{j,\ell'}$, and we define $\Lambda_{j,<\ell}, \Lambda_{j,\geq \ell}, \Lambda_{j,>\ell}$ similarly.

We will ensure that throughout the algorithm, the following invariant always holds:

I1. for all $1 \leq \ell \leq L_j$, $|\Lambda_{j,\geq \ell}| \leq n_{\leq j}/2^{\ell-1}$.

At the beginning of the algorithm, we set $\Lambda_{j,1} = \Lambda_j$ and $\Lambda_{j,\ell'} = \emptyset$ for all $1 < \ell' \leq L_j$. Consider now some sublayer $\Lambda_{j,\ell}$, for $\ell < L_j$. We partition the execution of our algorithm into phases with respect to this sublayer, that we refer to as *level-(j, ℓ) phases*, or *(j, ℓ)-phases*. Whenever Invariant I1 for the next sublayer $\Lambda_{j,\ell+1}$ is violated (that is, $|\Lambda_{j,\geq \ell+1}|$ exceeds $n_{\leq j}/2^{\ell}$), we terminate the current (j, ℓ)-phase and start a new phase.

Consider now some time t during the execution of the algorithm, when, for some $1 \leq \ell < L_j$, a (j, ℓ)-phase is terminated. Let ℓ' be the smallest index for which the (j, ℓ')-phase is terminated at time t. We then set $\Lambda_{j,\ell'} = \Lambda_{j,\geq \ell'}$, and for all $\ell' < \ell'' \leq L_j$, we set $\Lambda_{j,\ell''} = \emptyset$.

Throughout the algorithm, for every vertex u, we denote by $\deg_{\leq (j,\ell)}(u) = |E(u, \Lambda_{<j} \cup \Lambda_{j,\leq \ell})|$ the number of neighbors of u that lie in sublayer-(j, ℓ) or above (including in layers that are above layer j). By the definition, $\deg_{\leq (j,\ell)}(u) \leq \deg_{\leq j}(u)$. However, since, at the beginning of each (j, ℓ)-phase, we set $\Lambda_{j,\ell'} \leftarrow \emptyset$ for all $\ell' > \ell$, we obtain the following immediate observation:

Observation 3.12 *For all $1 \leq \ell < L_j$, for every vertex u, at the beginning of each (j, ℓ)-phase, $\deg_{\leq (j,\ell)}(u) = \deg_{\leq j}(u)$.*

Let $H_{j,\ell} = G[\Lambda_{j,\ell}]$ be the subgraph of G induced by the vertices of the sublayer $\Lambda_{j,\ell}$. We refer to $H_{j,\ell}$ as the *level-(j,ℓ) graph*. We will also ensure that throughout the algorithm the following invariant holds:

I2. For all $1 \leq \ell < L_j$, for each level-(j,ℓ) phase, graph $H_{j,\ell}$ only undergoes deletions of edges and vertices over the course of the phase.

Therefore, we say that graph $H_{j,\ell}$ is *decremental* within each (j,ℓ)-phase. Note that the graph H_{j,L_j} that corresponds to the buffer sublayer Λ_j^- may undergo both insertions and deletions of edges and vertices. As time progresses, some vertices v whose virtual degree $\widetilde{\deg}(v)$ was greater than h_j may have their virtual degree decrease to h_j. In order to preserve the above invariant, we always insert such vertices v into the buffer sublayer $\Lambda_j^- = \Lambda_{j,L_j}$; additional vertices may also be moved from higher sub-layers $\Lambda_{j,\ell}$ to the buffer sub-layer over the course of a (j,ℓ)-phase.

3.4 Initialization of a Sublayer: Core Decomposition

Consider now some non-buffer sub-layer $\Lambda_{j,\ell}$, with $\ell < L_j$. At the beginning of every (j,ℓ)-phase, if $\Lambda_{j,\ell} \neq \emptyset$, we compute a *core decomposition* of the graph $H_{j,\ell}$. This is one of the key subroutines in our LCD data structure. The following theorem provides the algorithm for computing the core decomposition of a sub-layer.

Theorem 3.13 (Core Decomposition of Sublayer) *There is a deterministic algorithm, that, given a level-(j,ℓ) graph $H_{j,\ell} = G[\Lambda_{j,\ell}]$, computes a collection $\mathcal{F}_{j,\ell}$ of vertex-disjoint subgraphs of $H_{j,\ell}$, called* cores*, such that each core $K \in \mathcal{F}_{j,\ell}$ is a φ-expander, and for every vertex $u \in V(K)$, $\deg_K(u) \geq \varphi \cdot \deg_{\leq(j,\ell)}(u)/12$. Moreover, if we denote by $U_{j,\ell} = \Lambda_{j,\ell} \setminus \left(\bigcup_{K\in\mathcal{F}_{j,\ell}} V(K)\right)$ the set of all vertices of $\Lambda_{j,\ell}$ that do not belong to any core, then there is an orientation of the edges of the graph $G[U_{j,\ell}]$, such that the resulting directed graph $\mathcal{D}_{j,\ell}$ is a directed acyclic graph (DAG), and, for every vertex $u \in U_{j,\ell}$, $\text{in-deg}_{\mathcal{D}_{j,\ell}}(u) \leq \deg_{\leq(j,\ell)}(u)/12$. The running time of the algorithm is $\widehat{O}(|E(H_{j,\ell})|)$.*

Proof: We use the following lemma, whose proof follows easily from Theorem 3.6.

Lemma 3.14 *There is a deterministic algorithm, that given a subgraph $H'_{j,\ell} \subseteq H_{j,\ell}$, such that every vertex $u \in V(H'_{j,\ell})$ has degree at least $\deg_{\leq(j,\ell)}(u)/12$ in $H'_{j,\ell}$, in time $\widehat{O}(|E(H_{j,\ell})|)$ computes a collection $\mathcal{F}$ of vertex-disjoint subgraphs of $H'_{j,\ell}$ called cores, such that each core $K \in \mathcal{F}$ a φ-expander and, for every vertex $u \in V(K)$, $\deg_K(u) \geq \varphi \deg_{\leq(j,\ell)}(u)/12$. Moreover, $\sum_{K\in\mathcal{F}} |E(K)| \geq 3|E(H'_{j,\ell})|/4$.*

Proof: We apply Theorem 3.6 to every connected component of graph $H'_{j,\ell}$, with parameter φ. Let $(V_1, \ldots, V_k)$ be the resulting partition of $V(H'_{j,\ell})$. For each $1 \leq i \leq k$, we the define a core $K_i = H'_{j,\ell}[V_i]$. Observe first that $\sum_{i=1}^k \delta(V_i) \leq \gamma \cdot \varphi |E(H'_{j,\ell})| \leq |E(H'_{j,\ell})|/4$ by our choice of φ. Therefore, $\sum_{i=1}^k |E(K_i)| \geq 3|E(H'_{j,\ell})|/4$. We are also guaranteed that for all $1 \leq i \leq k$, graph K_i is a strong φ-expander with respect to $H'_{j,\ell}$. Lastly, if K_i contains more than one vertex, then, from Observation 2.1, every vertex u of K_i has degree at least $\varphi \deg_{H'_{j,\ell}}(u) \geq \varphi \deg_{\leq(j,\ell)}(u)/12$ in K_i. We return a set $\mathcal{F}$ containing all graphs K_i with $|V(K_i)| > 1$. The running time of the algorithm is $\widehat{O}(|E(H'_{j,\ell})|)$ by Theorem 3.6. □

We are now ready to complete the proof of Theorem 3.13. Our algorithm is iterative. We start with $\mathcal{F}_{j,\ell} \leftarrow \emptyset$ and $H'_{j,\ell} \leftarrow H_{j,\ell}$. Consider the following *trimming* process similar to the one in [KT19]: while there is a vertex $u \in V(H'_{j,\ell})$ with $\deg_{H'_{j,\ell}}(u) < \deg_{\leq(j,\ell)}(u)/12$, delete u from $H'_{j,\ell}$. We say that

graph $H'_{j,\ell}$ is *trimmed* if, for all $u \in V(H'_{j,\ell})$ $\deg_{H'_{j,\ell}}(u) \geq \deg_{\leq(j,\ell)}(u)/12$. While $H'_{j,\ell} \neq \emptyset$, we perform iterations, each of which consists of the following steps:

1. Trim the current graph $H'_{j,\ell}$;
2. Apply the algorithm from Lemma 3.14 to graph $H'_{j,\ell}$, to obtain a collection $\mathcal{F}$ of cores;
3. For all $K \in \mathcal{F}$, delete all vertices of K from $H'_{j,\ell}$;
4. Set $\mathcal{F}_{j,\ell} \leftarrow \mathcal{F}_{j,\ell} \cup \mathcal{F}$.

Note that, throughout the algorithm, the graph $H'_{j,\ell}$ that serves as input to Lemma 3.14 has vertex degrees at least $\deg_{\leq(j,\ell)}(u)/12$ due to the trimming operation, so it is a valid input to the lemma. It is also immediate to see that the number of iterations is bounded by $O(\log n)$. Indeed, let H be the graph $H'_{j,\ell}$ at the beginning of some iteration, and let H' be the graph $H'_{j,\ell}$ at the beginning of the next iteration. Note that H' is a subgraph of $H \setminus (\bigcup_{K\in\mathcal{F}} E(K))$. As $|\bigcup_{K\in\mathcal{F}} E(K)| \geq 3|E(H)|/4$, we conclude that $|E(H')| \leq |E(H)|/4$. Therefore, after $O(\log n)$ iterations, $H'_{j,\ell}$ becomes empty and the algorithm terminates. It is easy to see that the trimming step takes time $O(|E(H'_{j,\ell})|)$. Therefore, the total running time of the algorithm is $\widehat{O}(|E(H_{j,\ell})|)$.

Consider now the final set $\mathcal{F}_{j,\ell}$ of cores computed by the algorithm. We now show that it has all required properties. From Lemma 3.14, we are guaranteed that each core $K \in \mathcal{F}_{j,\ell}$ is a φ-expander, and that for every vertex $u \in V(K)$, $\deg_K(u) \geq \varphi \cdot \deg_{\leq(j,\ell)}(u)/12$. Observe that every vertex in set $U_{j,\ell} = V(H_{j,\ell} \setminus \bigcup_{K\in\mathcal{F}_{j,\ell}} K)$ was deleted from graph $H'_{j,\ell}$ by the trimming procedure at some time t in the algorithm's execution. Therefore, at time t, $\deg_{H'_{j,\ell}}(u) < \deg_{\leq(j,\ell)}(u)/12$ held. We orient all edges that belonged to graph $H'_{j,\ell}$ at time t and are incident to u towards u. This provides an orientation of all edges in graph $G[U_{j,\ell}]$, which in turn defines a directed graph $\mathcal{D}_{j,\ell}$. From the above discussion, for every vertex of $\mathcal{D}_{j,\ell}$, $\text{in-deg}_{\mathcal{D}_{j,\ell}}(u) < \deg_{\leq(j,\ell)}(u)/12$ holds. Moreover, it is easy to see that graph $\mathcal{D}_{j,\ell}$ is a DAG, because the order in which the vertices of $U_{j,\ell}$ were deleted from $H'_{j,\ell}$ by the trimming procedure defines a valid topological ordering of the vertices of $\mathcal{D}_{j,\ell}$. □

3.5 Maintaining Cores and Supporting Short-Core-Path Queries

In this subsection, we describe an algorithm for maintaining the cores, and for supporting queries 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 $(\gamma(n))^{O(q)} = \widehat{O}(1)$, in time $(\gamma(n))^{O(q)} = \widehat{O}(1)$.

When we invoke the algorithm from Theorem 3.13 for computing a core decomposition of sublayer $\Lambda_{j,\ell}$ at the beginning of a (j,ℓ)-phase, we say that the core decomposition *creates* the cores in the set $\mathcal{F}_{j,\ell}$ that it computes. Our algorithm only creates new cores through the algorithm from Theorem 3.13, which may only be invoked at the beginning of a (j,ℓ)-phase.

Throughout the algorithm, we denote $\mathcal{F}_j = \mathcal{F}_{j,1} \cup \cdots \cup \mathcal{F}_{j,L_j-1}$, and we refer to graphs in $\mathcal{F}_j$ as *cores for layer* Λ_j, or *cores for graph* H_j (recall that we have defined $H_j = G[\Lambda_j]$). For convenience, we also use shorthand notation $\mathcal{F}_{j,\leq\ell} = \mathcal{F}_{j,1} \cup \cdots \cup \mathcal{F}_{j,\ell}$ and $\mathcal{F}_{\leq j} = \mathcal{F}_1 \cup \cdots \cup \mathcal{F}_j$. We define $\mathcal{F}_{\geq j}$ and $\mathcal{F}_{j,\geq\ell}$ analogously. Let $\hat{K}_j = \bigcup_{K\in\mathcal{F}_j} K$, and denote $\hat{K}_{\leq j} = \bigcup_{K\in\mathcal{F}_{\leq j}} K$. We define notation $\hat{K}_{\geq j}, \hat{K}_{j,\leq\ell}$ and $\hat{K}_{j,\geq\ell}$ analogously.

In order to maintain the cores and to support the Short-Core-Path(K, u, v) queries for each such core K, we do the following. Consider a pair $1 \leq j \leq r$, $1 \leq \ell < L_j$ of indices, and some core $K \in \mathcal{F}_{j,\ell}$, that was created when the core decomposition algorithm from Theorem 3.13 was invoked for layer $\Lambda_{j,\ell}$, at

the beginning of some (j,ℓ)-phase. Let $K^{(0)}$ denote the core K right after it is created, before any edges are deleted from K; recall that $K^{(0)}$ is a φ-expander. We use the algorithm from Theorem 3.9 on graph $K^{(0)}$, as it undergoes edge deletions, with the parameter q that serves as input to Theorem 3.4, to maintain the vertex set $S^K \subseteq V(K^{(0)})$. Whenever, over the course of the current (j,ℓ)-phase, an edge is deleted from graph G that belongs to $K^{(0)}$, we add this edge to the sequence of edge deletions from graph $K^{(0)}$, and update the set S^K of vertices using the algorithm from Theorem 3.9 accordingly. At any point in the current (j,ℓ)-phase, if $A^K \subseteq E(K^{(0)})$ is the set of edges of $K^{(0)}$ that were deleted from G so far, and S^K is the current vertex set maintained by the algorithm from Theorem 3.9, then we set the current core corresponding to $K^{(0)}$ to be the graph obtained from K^0 by deleting the edges of A^K and the vertices of S^K from it; in other words, $K = (K^{(0)} \setminus A^K) \setminus S^k$. We refer to the resulting graph K as a core throughout the phase. Whenever the number of deleted edges in A^K exceeds $\varphi|E(K^{(0)})|/10$, we set $S^K = V(K^{(0)})$, which effectively set $K = \emptyset$; at this point we say that core K is *destroyed*. Each destroyed core is removed from $\mathcal{F}_{j,\ell}$.

From this definition of the core K, from the time it is created and until it is destroyed, it may only undergo deletions of edges and vertices. In addition to the deletion of edges of K due to the edge deletions from the input graph G, we also delete vertices of S^K from K. Whenever a vertex $v \in V(K)$ is deleted from K (that is, v is added to S^K), we say that v is *pruned out* of K. When there are more than $\varphi|E(K^{(0)})|/10$ edge deletions in A^K, all vertices of K are pruned out and so K is destroyed.

Therefore, we can now use Theorem 3.9 in order to support queries $\mathsf{Short\text{-}Core\text{-}Path}(K,u,v)$ for each core K: given a pair $u,v \in V(K)$ of vertices of K, return a simple u-v path P in K of length at most $(\gamma(m))^{O(q)}$ in time $(\gamma(m))^{O(q)}$. We now provide a simple observation about the maintained cores.

Proposition 3.15 *For every core K, from the time K is created and until it is destroyed, $|V(K)| \geq \Omega(\varphi^2 h_j)$ holds.*

Proof: By Theorem 3.9, K is a strong $\varphi/6$-expander w.r.t. $K^{(0)}$. Let $u \in V(K)$ be a vertex minimizing $\deg_{K^{(0)}}(u)$. In particular, $\deg_{K^{(0)}}(u) \leq \mathrm{vol}_{K^{(0)}}(V(K))/2$ must hold. By Observation 2.2, $\deg_K(u) \geq (\varphi/6) \cdot \deg_{K^{(0)}}(u)$. Since, at the beginning of the (j,ℓ)-phase

$$\begin{aligned}
\deg_{K^{(0)}}(u) &\geq \varphi \deg_{\leq (j,\ell)}(u)/12 && \text{by Theorem 3.13} \\
&= \varphi \deg_{\leq j}(u)/12 && \text{by Observation 3.12} \\
&\geq \varphi h_j/12 && \text{by Observation 3.1,}
\end{aligned}$$

held we conclude that $|V(K)| \geq \deg_K(u) = \Omega(\varphi^2 h_j)$ (using the fact that the graph is simple). □

We use the following observation in order to bound the number of cores *at any point during the algorithm's execution*. Later in Section 3.7, we will give another bound for the total number of cores *ever created* by the algorithm.

Observation 3.16 *For all $1 \leq j \leq r$ and $1 \leq \ell < L_j$, at any time over the course of the algorithm, $|\mathcal{F}_{j,\ell}| \leq O(|\Lambda_{j,\ell}|/(\varphi^2 h_j))$, and $|\mathcal{F}_{\leq j}| \leq O(n_{\leq j}/(\varphi^2 h_j))$ must hold. Moreover, if C is a connected component of $G[\Lambda_{\leq j}]$, and $\mathcal{F}^C_{\leq j} = \{K \in \mathcal{F}_{\leq j} \mid K \subseteq C\}$ is the collection of cores in $\mathcal{F}_{\leq j}$ that are contained in C, then $|\mathcal{F}^C_{\leq j}| \leq O(|V(C)|/(\varphi^2 h_j))$.*

Proof: Consider a set $\mathcal{F}_{j,\ell}$ of remaining cores in sublayer $\Lambda_{j,\ell}$ which is not destroyed yet. Note again that new cores in $\mathcal{F}_{j,\ell}$ may only be created when the algorithm from Theorem 3.13 is employed on sublayer $\Lambda_{j,\ell}$. In the beginning of a (j,ℓ)-phase, all cores are vertex disjoint by Theorem 3.13. Moreover, each core undergoes deletions only so it remains disjoint and it contains $\Omega(\varphi^2 h_j)$ vertices at any point of time before it is destroyed by Proposition 3.15. So we the number of cores is at most

$|\mathcal{F}_{j,\ell}| = O(|\Lambda_{j,\ell}|/(\varphi^2 h_j))$ at any point of time. By summing up the above bound over all sublayers in layers $1,\ldots,j$, and noting that $h_1, h_2, \ldots, h_j$ form a geometrically decreasing sequence, and that $|\Lambda_{\leq j}| \leq n_{\leq j}$ holds at all times, we get that $|\mathcal{F}_{\leq j}| \leq O(n_{\leq j}/(\varphi^2 h_j))$.

Lastly, consider some connected component C of $G[\Lambda_{\leq j}]$, and let $\mathcal{F}^C_{\leq j} = \{K \in \mathcal{F}_{\leq j} \mid K \subseteq C\}$. For an index $1 \leq \ell < L_j$, let $\mathcal{F}^C_{j,\ell} = \{K \in \mathcal{F}_{j,\ell} \mid K \subseteq C\}$. Using the same argument, $|\mathcal{F}^C_{j,\ell}| \leq O(|\Lambda_{j,\ell} \cap V(C)|/(\varphi^2 h_j))$. By summing up over all sub-layers of layers $1,\ldots,j$, we conclude that $|\mathcal{F}^C_{\leq j}| \leq O(|V(C)|/(\varphi^2 h_j))$. □

3.6 Maintaining the Structure of the Sublayers

In this subsection we provide additional details regarding the sub-layers of each layer Λ_j, and in particular we describe how vertices move between the sublayers. Throughout this subsection, we fix an index $1 \leq j \leq r$.

Consider an index $1 \leq \ell < L_j$. Throughout the algorithm, we maintain a partition of the vertices of the (non-buffer) sub-layer $\Lambda_{j,\ell}$ into two subsets: set $\hat{K}_{j,\ell}$ that contains all vertices currently lying in the cores of $\mathcal{F}_{j,\ell}$, so $\hat{K}_{j,\ell} = \bigcup_{K \in \mathcal{F}_{j,\ell}} V(K)$, and set $U_{j,\ell}$ containing all remaining vertices of $\Lambda_{j,\ell}$. (We note that previously, we defined $\hat{K}_{j,\ell}$ to denote the graph $\bigcup_{K \in \mathcal{F}_{j,\ell}} K$; we slightly abuse the notation here by letting $\hat{K}_{j,\ell}$ denote the set of vertices of this graph). For every vertex $u \in \Lambda_{j,\ell}$, let $\deg^{(0)}_{\leq (j,\ell)}(u)$ and $\deg^{(0)}_{\leq j}(u)$ denote $\deg_{\leq (j,\ell)}(u)$ and $\deg_{\leq j}(u)$ at the beginning of the current (j,ℓ)-phase, respectively. Recall that, from Observation 3.12, $\deg^{(0)}_{\leq (j,\ell)}(u) = \deg^{(0)}_{\leq j}(u)$. We maintain the following invariant:

I3. For every vertex $u \in U_{j,\ell}$, $\deg_{\leq (j,\ell)}(u) \geq \deg^{(0)}_{\leq (j,\ell)}(u)/4$ holds throughout the execution of a (j,ℓ)-phase.

We now consider the buffer sublayer Λ^-_j. The vertices of the buffer sublayer are partitioned into three disjoint subsets: $\hat{K}^-_j$, U^-_j, and D^-_j, that are defined as follows. First, for all $1 \leq \ell < L_j$, whenever any vertex u is pruned out of any core $K \in \mathcal{F}_{j,\ell}$ by the core pruning algorithm from Theorem 3.7 over the course of the current (j,ℓ)-phase, vertex u is deleted from sublayer $\Lambda_{j,\ell}$ and is added to the buffer sublayer Λ^-_j, where it joins the set $\hat{K}^-_j$ (recall that, once the current (j,ℓ)-phase terminates, we set $\Lambda^-_j = \emptyset$). Additionally, whenever Invariant I3 is violated for any vertex $u \in U_{j,\ell}$, we delete u from $\Lambda_{j,\ell}$, and add it to Λ^-_j, where it joins the set U^-_j. Lastly, for all $j' < j$, whenever a vertex $u \in \Lambda_{j'}$ has its virtual degree decrease from $h_{j'}$ to h_j, vertex u is added to layer Λ_j, into the buffer sublayer Λ^-_j, where it joins the set D^-_j. Similarly, whenever a vertex $u \in \Lambda_j$ has its virtual degree decrease below h_j, we delete it from Λ_j and move it to the appropriate layer, where it joints the corresponding buffer sub-layer.

Whenever a vertex is added to the buffer sublayer Λ^-_j, we say that a *move into* Λ^-_j occurs. The following lemma, that is key to the analysis of the algorithm, bounds the number of such moves. Recall that we used $n_{\leq j}$ to denote the number of vertices in $\Lambda_{\leq j}$ at the beginning of the algorithm, before any edges are deleted from G.

Lemma 3.17 *For all $1 \leq j \leq r$, the total number of moves into Λ^-_j over the course of the entire algorithm is at most $O(n_{\leq j}\Delta/\varphi^3) = \widehat{O}(n_{\leq j}\Delta)$.*

We defer the proof of the lemma to Section 3.8, after we show, in Section 3.7, several immediate useful consequences of the lemma.

3.7 Bounding the Number of Phases and the Number of Cores

In this subsection we use Lemma 3.17 to bound the number of phases and the number of cores for each sublayer $\Lambda_{j,\ell}$. Recall that in Section 3.5 we have described an algorithm for maintaining each core $K \in \mathcal{F}_{j,\ell}$ over the course of a (j,ℓ)-phase. The set $\mathcal{F}_{j,\ell}$ of cores is initialized using the Core Decomposition algorithm from Theorem 3.13, at the beginning of a (j,ℓ)-phase. After that, every core $K \in \mathcal{F}_{j,\ell}$ only undergoes edge and vertex deletions, over the course of the (j,ℓ)-phase. Once $K = \emptyset$ holds, or a new (j,ℓ)-phase starts, we say that the core is *destroyed*. Recall that $\mathcal{F}_j = \mathcal{F}_{j,1} \cup \cdots \cup \mathcal{F}_{j,L_j-1}$ denotes the set of all cores in Λ_j (we do not perform core decomposition in the buffer sublayer Λ_{j,L_j}). In this section, using Lemma 3.17, we bound the total number of cores in Λ_j that are ever created over the course of the algorithm, and also the number of (j,ℓ)-phases, for all $1 \leq \ell < L_j$, in the next two lemmas.

Lemma 3.18 *For all $1 \leq j \leq r$ and $1 \leq \ell < L_j$, the total number of (j,ℓ)-phases over the course of the algorithm is at most $\widehat{O}(2^\ell \Delta)$.*

Proof: Fix a pair of indices $1 \leq j \leq r$, $1 \leq \ell < L_j$. Recall that we start a new (j,ℓ)-phase only when Invariant I1 is violated for sublayer $(j,\ell+1)$, that is, $|\Lambda_{j,\geq \ell+1}| > n_{\leq j}/2^\ell$ holds. At the beginning of a (j,ℓ)-phase, we set $\Lambda_{j,\ell'} = \emptyset$ for all $\ell' > \ell$, and so in particular, $\Lambda_{j,\geq \ell+1} = \emptyset$. The only way that new vertices are added to set $\Lambda_{j,\geq \ell+1}$ is when new vertices join the buffer layer Λ_j^-, that is, via a move into the buffer sublayer. Therefore, at least $n_{\leq j}/2^\ell$ moves into the buffer sublayer Λ_j^- are required before the current (j,ℓ)-phase terminates. Since, from Lemma 3.17, the total number of moves into Λ_j^- is bounded by $\widehat{O}(n_{\leq j}\Delta)$, the total number of (j,ℓ)-phases is bounded by $\widehat{O}(2^\ell \Delta)$. □

Lastly, we bound the total number of cores in $\mathcal{F}_j$ that are ever created over the course of the algorithm in the following lemma.

Lemma 3.19 *The total number of cores created in layer Λ_j over the course of the entire algorithm is at most $\widehat{O}(n_{\leq j} \cdot \Delta/h_j)$.*

Proof: Consider an index $1 \leq \ell < L$. By Observation 3.16, at the beginning of every (j,ℓ)-phase, $|\mathcal{F}_{j,\ell}| \leq O(|\Lambda_{j,\ell}|/(\varphi^2 h_j)) = \widehat{O}(n_{\leq j}/(2^\ell h_j))$ holds (we have used Invariant I1 to bound $|\Lambda_{j,\ell}|$ by $n_{\leq j}/2^{\ell-1}$). Since the total number of (j,ℓ)-phases over the course of the algorithm is bounded by $\widehat{O}(2^\ell \Delta)$, the total number of cores that are ever created in sublayer $\Lambda_{j,\ell}$ is bounded by $\widehat{O}(n_{\leq j}\Delta/h_j)$. The claim follows by summing over all $L_j - 1 = O(\log n)$ sublayers. □

3.8 Bounding the Number of Moves into the Buffer Sublayers: Proof of Lemma 3.17

The goal of this subsection is to prove Lemma 3.17. Throughout this subsection, we fix an index $1 \leq j \leq r$. Our goal is to prove that the total number of moves into the buffer sublayer Λ_j^- over the course of the entire algorithm is bounded by $\widehat{O}(n_{\leq j}\Delta)$. We partition all moves into the buffer sublayer Λ_j^- into three types. A move of a vertex u into sublayer Λ_j^- is of *type-D*, if u is added to set D_j^-; it is of type-K, if u is added to K_j^-; and it is of type-U if u is added to set U_j^-. We now bound the number of moves of each type separately.

Type-D Moves. Recall that a vertex u is added to D_j^- only if its virtual degree decreases from some value h'_j for $j' < j$ to h_j. Since virtual degrees of all vertices only decrease, such a vertex must lie in $\Lambda_{\leq j}$ at the beginning of the algorithm, and each such vertex u may only be added to set D_j^- once

over the course of the algorithm. Therefore, the total number of type-D moves into Λ_j^- is bounded by $n_{\leq j}$.

Type-K Moves. To bound the number of type-K moves, it is convenient to assign *types* to edge deletions. Consider an index $1 \leq \ell < L_j$ and the corresponding graph $H_{j,\ell} = G[\Lambda_{j,\ell}]$. Let e be an edge deleted from $H_{j,\ell}$. We assign to the edge e one of the following four deletion types.

- If e is deleted by the adversary (that is, e is deleted as part of the deletion sequence of the input graph G), then this deletion is of *type-A*;
- If e is deleted from $H_{j,\ell}$ because the virtual degree of one of its endpoints falls below h_j (and so that endpoint is deleted from Λ_j), then this deletion is of *type-D*;
- If an endpoint of e belonged to some core $K \in \mathcal{F}_{j,\ell}$, and is then pruned out of that core (and so that endpoint is removed from $\Lambda_{j,\ell}$ and added to K_j^-), then the deletion of edge e is of *type-K*;
- Lastly, if an endpoint u of e lies in $u \in U_{j,\ell}$, and Invariant I3 stops holding for u, that is, $\deg_{\leq(j,\ell)}(u) < \deg^{(0)}_{\leq(j,\ell)}(u)/4$ holds (and so u is removed from $\Lambda_{j,\ell}$ and added to U_j^-), then the deletion of e is of *type-U*.

Observe that every edge deletion from a graph $H_{j,\ell}$ executed over the course of a (j,ℓ)-phase must fall under one of these four categories. As the algorithm progresses, the same vertex may be added to and deleted from sublayer $\Lambda_{j,\ell}$ multiple times. Therefore, an edge that is deleted from $H_{j,\ell}$ may be re-added to the new graph $H_{j,\ell}$ at the beginning of one of the subsequent (j,ℓ) phases. Next, we bound the total number of edge deletions from all graphs $H_{j,\ell}$ over the course of the entire algorithm, for the first three types. Notice that these edge deletions ignore the deletions of edges whose endpoints belong to different sublayers.

The following simple observation bounds the number of type-A and type-D deletions.

Observation 3.20 *The total number of type-A and type-D deletions from all graphs $H_{j,\ell}$ for all $1 \leq \ell < L_j$, over all (j,ℓ)-phases is bounded by $n_{\leq j}h_j\Delta$.*

Proof: Observe that each type-A deletion corresponds to a deletion of an edge whose both endpoints are contained in Λ_j from the input graph G; each such edge may only be deleted once over the course of the algorithm. From Observation 3.3, the total number of such edges is bounded by $n_{\leq j}h_j\Delta$.

If an edge e is deleted in a type-D deletion from some graph $H_{j,\ell}$, then both its endpoints lie in Λ_j, and, after the deletion, one of the endpoints of e is removed from layer Λ_j forever. Therefore, every edge may be deleted at most once in a type-D deletion, and the number of all such deletions is bounded by the total number of edges whose both endpoints are contained in Λ_j over the course of the algorithm, which is again bounded by $n_{\leq j}h_j\Delta$ from Observation 3.3. □

We now proceed to bound the total number of type-K edge deletions.

Lemma 3.21 *The total numbers of type-K edge deletions from all graphs $H_{j,\ell}$ for all $1 \leq \ell < L_j$, over all (j,ℓ)-phases is bounded by $O(n_{\leq j}h_j\Delta/\varphi^2)$.*

Proof: Consider an index $1 \leq \ell < L_j$, and some (j,ℓ)-phase. Let $H^{(0)}_{j,\ell}$ denote the graph $H_{j,\ell}$ at the beginning of the (j,ℓ)-phase, and let $K^{(0)}$ denote an arbitrary core $K \in \mathcal{F}_{j,\ell}$ at the beginning of that phase. Let S^K be the set of vertices that are pruned out of $K^{(0)}$ by Theorem 3.9. By the definition

of type-K deletion, it is enough to bound $\mathrm{vol}_{H^{(0)}_{j,\ell}}(S^K)$, summing over cores K created in $H_{j,\ell}$ over the course of the algorithm.

In order to bound $\mathrm{vol}_{H^{(0)}_{j,\ell}}(S^K)$ for a single core $K \in \mathcal{F}_{j,\ell}$, recall that, from Theorem 3.13, for every vertex $u \in V(K^{(0)})$ $\deg_{K^{(0)}}(u) \geq \varphi \deg_{H^{(0)}_{j,\ell}}(u)/12$ holds. Therefore, $\mathrm{vol}_{H^{(0)}_{j,\ell}}(S^K) \leq 12 \cdot \mathrm{vol}_{K^{(0)}}(S^K)/\varphi$. Moreover, by Theorem 3.9, after t edge deletions from $K^{(0)}$ (that include type-A and type-D deletions, but exclude type-U deletions, as edges deleted this way must lie outside of the core), we have $\mathrm{vol}_{K^{(0)}}(S^K) \leq O(t/\varphi)$. From Observation 3.20, the total number of type-A and type-D edge deletions, in all graphs $H_{j,\ell}$, for all $1 \leq \ell < L_j$ and all (j,ℓ)-phases, is at most $n_{\leq j} h_j \Delta$. We conclude that the sum of all volumes $\mathrm{vol}_{H^{(0)}_{j,\ell}}(S^K)$, over all $1 \leq \ell < L_j$, and over all cores ever created in $\mathcal{F}_{j,\ell}$, is bounded by: $\frac{12}{\varphi} \cdot O(n_{\leq j} h_j \Delta/\varphi) = O(n_{\leq j} h_j \Delta/\varphi^2)$. □

Corollary 3.22 *The total number of type-K moves into Λ_j^- over the course of the algorithm is bounded by $O(n_{\leq j}\Delta/\varphi^3)$.*

Proof: Consider some sublayer $\Lambda_{\ell,j}$, and the graph $H^{(0)}_{j,\ell}$ at the beginning of some (j,ℓ)-phase. Let $K^{(0)} \in \mathcal{F}_{j,\ell}$ be some core that was created at the beginning of that phase, and let $u \in V(K^{(0)})$ be any vertex of the core. Recall that, from Theorem 3.13, $\deg_{K^{(0)}}(u) \geq \varphi \deg^{(0)}_{\leq(j,\ell)}(u)/12$, and from Observation 3.12, $\deg^{(0)}_{\leq(j,\ell)}(u) = \deg^{(0)}_{\leq j}(u) \geq h_j$. Therefore, $\deg_{K^{(0)}}(u) \geq \varphi h_j/12$. If vertex u is moved to Λ_j^- via a type-K move, then each of the $\Omega(\varphi h_j)$ edges of $K^{(0)}$ incident to u must have been deleted as part of A, D, or K-type deletion from $H_{j,\ell}$. Since the total number of all deletions of types A, D and K, from all graphs $H_{j,\ell}$ for $1 \leq \ell < L_j$, over the course of the whole algorithm, is bounded by $O(n_{\leq j} h_j \Delta/\varphi^2)$, we get that the total number of K-type moves into the buffer layer Λ_j^- over the course of the entire algorithm is bounded by $O(n_{\leq j}\Delta/\varphi^3)$. □

Type-U Moves. We further partition type-U moves into two subtypes. Consider a time in the algorithm's execution, when some vertex u is added to set U_j^-, so it is added to Λ_j^- via a U-move, and assume that u was moved from sublayer $\Lambda_{j,\ell}$. We say that this move is of type-U_1 if $|E_G(u, \Lambda_{j,>\ell})| < 2\deg_{\leq(j,\ell)}(u)$ held right before u is moved, and we say that it is of type-U_2 otherwise. We bound the number of moves of both subtypes separately.

We first bound the number of type-U_1 moves. Recall that, whenever u is type-U_1 moved:

$$\begin{aligned}\deg_{\leq j}(u) &= \deg_{\leq(j,\ell)}(u) + |E_G(u, \Lambda_{j,>\ell})| \\ &< 3\deg_{\leq(j,\ell)}(u) \\ &< 3\deg^{(0)}_{\leq(j,\ell)}(u)/4 \\ &= 3\deg^{(0)}_{\leq j}(u)/4\end{aligned}$$

(we have used the fact that Invariant I3 is violated when u is moved to Λ_j^-, and Observation 3.12 for the last equality.) Therefore, the number of neighbors of u in $\Lambda_{\leq j}$ has reduced by a constant factor compared to the beginning of the current (j,ℓ)-phase. Note that $\deg_{\leq j}(u)$ may never increase, as virtual degrees of vertices may only decrease. Therefore, a vertex may be moved to Λ_j^- via a type-U_1 move at most $O(\log n)$ times. The total number of type-U_1 moves into Λ_j^- over the course of the entire algorithm is then bounded by $O(n_{\leq j} \log n)$.

It remains to bound the the total number of type-U_2 moves into Λ_j^-. We will show that the total number of such moves is bounded by $O(n_{\leq j}\Delta/\varphi)$ over the course of the algorithm. For all $1 \leq \ell < L_j$,

we define an edge set $\Pi_{j,\ell}$, that contains all edges $e=(u,v)$ with $u \in \Lambda_{j,\ell}$ and $v \in \Lambda_{j,>\ell}$. Intuitively, set $\Pi_{j,\ell}$ contains all "downward edges" from vertices in $\Lambda_{j,\ell}$ that lie *within the layer* j. Note that $\Pi_{j,L_j}=\emptyset$. We first bound the total number of edges that ever belonged to each such set $\Pi_{j,\ell}$ over the course of the algorithm. (Note that an edge may be added several times to $\Pi_{j,\ell}$ over the course of the algorithm; we count them as separate edges). We will then use this bound in order to bound the total number of the U_2-moves.

Note that a new edge $e=(u,v)$ may only be added to set $\Pi_{j,\ell}$ in the following cases:

- **(Type-D addition):** when some vertex $v \in \Lambda_{<j}$ is type-D moved to $D_j^- \subseteq \Lambda_j^-$. We denote the set of all edges added to $\Pi_{j,\ell}$ in a type-D addition by $\Pi_{j,\ell}^D$.
- **(Type-K addition):** when some vertex $v \in \Lambda_{j,\leq \ell}$ is type-K moved to $K_j^- \subseteq \Lambda_j^-$. We denote the set of all edges added to $\Pi_{j,\ell}$ in a type-K addition by $\Pi_{j,\ell}^K$.
- **(Type-U_1 addition):** when some vertex $v \in \Lambda_{j,\leq \ell}$ is type-U_1 moved to $U_j^- \subseteq \Lambda_j^-$. We denote the set of all edges added to $\Pi_{j,\ell}$ in a type-U_1 addition by $\Pi_{j,\ell}^{U_1}$.
- **(Type-U_2 addition):** when some vertex $v \in \Lambda_{j,\leq \ell}$ is type-U_2 moved to $U_j^- \subseteq \Lambda_j^-$. We denote the set of all edges added to $\Pi_{j,\ell}$ in a type-U_2 addition by $\Pi_{j,\ell}^{U_2}$.

Observe that the core decomposition algorithm from Theorem 3.13 that is performed at the beginning of a (j,ℓ)-phase does not add any new edges to any set $\Pi_{j,\ell'}$, since, for $\ell' > \ell$, we set $\Lambda_{j,\ell'}=\emptyset$, and for $\ell' \leq \ell$, vertex set $\Lambda_{j,\geq \ell'}$ remains unchanged.

Let $\mathtt{inc}_{j,\ell}$ denote the total number of edges ever added to $\Pi_{j,\ell}$, and for every addition type $X \in \{D,K,U_1,U_2\}$, we denote by $\mathtt{inc}_{j,\ell}^X$ the total number of edges ever added to $\Pi_{j,\ell}^X$. Clearly, $\mathtt{inc}_{j,\ell} = \sum_{X\in\{D,K,U_1,U_2\}} \mathtt{inc}_{j,\ell}^X$. For convenience, for all $X \in \{D,K,U_1,U_2\}$, we denote by $\Pi_j^X = \bigcup_{1\leq \ell<L_j} \Pi_{j,\ell}^X$. Let $\mathtt{inc}_j$ denote the total number of edges ever added to any of the sets in $\{\Pi_{j,\ell} \mid 1 \leq \ell < L_j\}$. Similarly, for all $X \in \{D,K,U_1,U_2\}$, we denote by $\mathtt{inc}_j^X$ the total number of edges ever added to any of the sets in $\left\{\Pi_{j,\ell}^X \mid 1 \leq \ell < L_j\right\}$. Observe that $\mathtt{inc}_j = \sum_{X\in\{D,K,U_1,U_2\}} \mathtt{inc}_j^X$. We start by bounding $\mathtt{inc}_j^D$, $\mathtt{inc}_j^{U_1}$, and $\mathtt{inc}_j^K$:

Lemma 3.23 *$\mathtt{inc}_j^D \leq O(n_{\leq j}h_j\Delta)$, $\mathtt{inc}_j^{U_1} \leq O(n_{\leq j}h_j \log n)$ and $\mathtt{inc}_j^K \leq O(n_{\leq j}h_j\Delta/\varphi^2)$.*

Proof: When a vertex u is moved to Λ_j^-, its contribution to $\mathtt{inc}_j$ is at most $|E(u,\Lambda_j)|$. By Observation 3.3, the total number of edges e, such that, at any point during the algorithm's execution, both endpoints of e were contained Λ_j, is at most $n_{\leq j}h_j\Delta$. As each vertex u can moved to Λ_{j^-} in a type-D move only once, $\mathtt{inc}_j^D \leq O(n_{\leq j}h_j\Delta)$ must hold. Similarly, as each vertex u can be moved to Λ_j^- in a type-U_1 move at most $O(\log n)$ times, $\mathtt{inc}_j^{U_1} \leq O(n_{\leq j}h_j\Delta \log n)$ must hold.

Next, observe that $\mathtt{inc}_{j,\ell}^K$ is precisely the total number of type-K edge deletions from graphs $H_{j,1},\ldots,H_{j,\ell}$ over the course of the algorithm. By Lemma 3.21, we can bound $\mathtt{inc}_j^K \leq O(n_{\leq j}h_j\Delta/\varphi^2)$. □

We use the next lemma to bound $\mathtt{inc}_{j,\ell}^{U_2}$.

Lemma 3.24 *For all $1 \leq \ell < L_j$, $\mathtt{inc}_{j,\ell}^{U_2} \leq \mathtt{inc}_{j,\ell}^D + \mathtt{inc}_{j,\ell}^K + \mathtt{inc}_{j,\ell}^{U_1}$.*

Proof: Fix an index $1 \leq \ell < L_j$. We denote by $\hat{\Pi}_{j,\ell}$ the set of all edges that were ever present in set $\Pi_{j,\ell}$ over the course of the algorithm, and by $\hat{\Pi}_{j,\ell}^{U_2}$ the set of all edges that were ever present in set $\Pi_{j,\ell}^{U_2}$ over the course of the entire algorithm; recall that $\hat{\Pi}_{j,\ell}^{U_2} \subseteq \hat{\Pi}_{j,\ell}$. We next show that $|\hat{\Pi}_{j,\ell}^{U_2}| \leq |\hat{\Pi}_{j,\ell}|/2$.

In order to do so, we assign, to every edge $e \in \hat{\Pi}^{U_2}_{j,\ell}$, two edges $e^1, e^2 \in \hat{\Pi}_{j,\ell}$ that are *responsible* for e. We will ensure that every edge in $\hat{\Pi}_{j,\ell}$ is responsible for at most one edge in $\hat{\Pi}^{U_2}_{j,\ell}$. This will immediately imply that $|\hat{\Pi}^{U_2}_{j,\ell}| \leq |\hat{\Pi}_{j,\ell}|/2$.

Consider now some (j,ℓ)-phase, and some vertex $u \in \Lambda_{j,\ell}$, that is moved to Λ^-_j via a U_2-move some time during the (j,ℓ)-phase. At the beginning of the (j,ℓ)-phase, $\Lambda_{j,\ell'} = \emptyset$ held for all $\ell' > \ell$. At the time when u is moved to Λ^-_j, from the definition of a U_2-move, $|E_G(u, \Lambda_{j,>\ell})| \geq 2\deg_{\leq(j,\ell)}(u)$ held. The edges that are added to $\hat{\Pi}^{U_2}_{j,\ell}$ due to the move of u to Λ^-_j are the edges of $E_G(u, \Lambda_{j,\ell})$. On the other hand, each edge $(u,v) \in E_G(u, \Lambda_{j,>\ell})$ belonged to set $\Pi_{j,\ell}$ before the move of u, and is removed from that set afterwards. Moreover, vertex v must have been moved to Λ^-_j at some time during the course of the current (j,ℓ)-phase. Since $|E_G(u, \Lambda_{j,>\ell})| \geq 2\deg_{\leq(j,\ell)}(u) \geq 2\deg_{(j,\ell)}(u)$, we can select, for every edge $e \in E_G(u, \Lambda_{j,\ell})$ arbitrary two edges $e^1, e^2 \in E_G(u, \Lambda_{j,>\ell})$ that become responsible for e, such that every edge of $E_G(u, \Lambda_{j,>\ell})$ is responsible for at most one edge of $E_G(u, \Lambda_{j,\ell})$. It is clear from this process that every edge in $\hat{\Pi}_{j,\ell}$ is responsible for at most one edge in $\hat{\Pi}^{U_2}_{j,\ell}$. We conclude that $|\hat{\Pi}^{U_2}_{j,\ell}| \leq |\hat{\Pi}_{j,\ell}|/2$ holds, and so $\mathtt{inc}^{U_2}_{j,\ell} \leq \mathtt{inc}_{j,\ell}/2$. Since $\mathtt{inc}_{j,\ell} = \sum_{X \in \{D,K,U_1,U_2\}} \mathtt{inc}^X_{j,\ell}$, we get that $\mathtt{inc}^{U_2}_{j,\ell} \leq \mathtt{inc}^D_{j,\ell} + \mathtt{inc}^K_{j,\ell} + \mathtt{inc}^{U_1}_{j,\ell}$. □

Combining Lemma 3.23 and Lemma 3.24, we obtain the following corollary.

Corollary 3.25 $\mathtt{inc}_j \leq O(n_{\leq j} h_j \Delta/\varphi^3)$.

Lastly, the following corollary allows us to bound the total number of U_2-moves.

Corollary 3.26 *The total number of U_2-moves into the buffer layer Λ^-_j, over the course of the entire algorithm, is bounded by $O(n_{\leq j}\Delta/\varphi^3)$.*

Proof: Consider some index $1 \leq \ell < L_j$, some (j,ℓ)-phase, and some vertex $u \in \Lambda_{j,\ell}$, that is moved to Λ^-_j via a U_2-move some time during that (j,ℓ)-phase. From the definition of a U_2-move, when u was moved to Λ^-_j, $|E_G(u, \Lambda_{j,>\ell})| \geq 2\deg_{\leq(j,\ell)}(u)$ held. Moreover, each edge $(u,v) \in E_G(u, \Lambda_{j,>\ell})$ belonged to set $\Pi_{j,\ell}$ before the move of u, and is removed from that set afterwards. Since $|E_G(u, \Lambda_{j,>\ell})| \geq 2\deg_{\leq(j,\ell)}(u)$ and $|E_G(u, \Lambda_{j,>\ell})| + \deg_{\leq(j,\ell)}(u) = \deg_{\leq j}(u)$, we get that $|E_G(u, \Lambda_{j,>\ell})| \geq \deg_{\leq j}(u)/3$. From the definition of virtual degrees, $\deg_{\leq j}(u) \geq h_j$ must hold, and so $|E_G(u, \Lambda_{j,>\ell})| \geq h_j/3$. Therefore, for every vertex that is added to Λ^-_j via a U_2-move, we delete at least $h_j/3$ edges from set $\Pi_{j,\ell}$. Since, as shown above, the total number of edges that are ever added to sets $\Pi_{j,\ell}$, for all $1 \leq \ell < L_j$ is bounded by $O(n_{\leq j} h_j \Delta/\varphi^3)$, the total number of U_2-moves into Λ_j over the entire course of the algorithm is bounded by $O(n_{\leq j}\Delta/\varphi^3)$. □

To summarize, we have partitioned all moves into the buffer sublayer Λ^-_j into four types: D, A, K and U, and we showed that the total number of moves of each type, over the course of the algorithm, is bounded by $O(n_{\leq j}\Delta/\varphi^3)$. Therefore, the total number of moves into Λ^-_j over the course of the algorithm is at most $O(n_{\leq j}\Delta/\varphi^3) \leq \widehat{O}(n_{\leq j}\Delta)$.

3.9 Existence of Short Paths to the Cores

The main result of this subsection is summarized in the following lemma, which shows that, throughout the algorithm, for every vertex $v \in V(G)$, there is a path of length $O(\log^3 n)$, connecting v to some vertex that lies in one of the cores of $\bigcup_j \mathcal{F}_j$. This fact will be used to process To-Core-Path queries.

Lemma 3.27 *Throughout the algorithm, for each vertex $u \in V(G)$, there is a path P_u of length at most $O(\log^3 n)$, connecting u to a vertex v lying in set $\hat{K} = \bigcup_{j,\ell} \hat{K}_{j,\ell}$. Moreover, the path $P_u =$*

$\{u = u_1, u_2, \ldots, u_k\}$ *is non-decreasing with respect to the sublayers, that is, if* $u_i \in \Lambda_{j,\ell}$*, then* $u_{i+1} \in \Lambda_{<j} \cup \Lambda_{j,\leq \ell}$.

Proof: We use the following two claims.

Claim 3.28 *Throughout the algorithm, for all* $1 \leq j \leq r$*, every vertex* u *in the buffer sublayer* Λ_{j,L_j} *has a neighbor in* $\Lambda_{<j} \cup \Lambda_{j,<L_j}$.

Proof: Recall that Invariant I1 guarantees that $|\Lambda_{j,L_j}| \leq n_{\leq j}/2^{L_j-1}$, and, from the choice of L_j, $n_{\leq j}/2^{L_j-1} \leq h_j/2$. From the definition of virtual degrees of vertices, for every vertex $u \in \Lambda_j^-$, $|E_G(u, \Lambda_{\leq j}) \geq h_j$ must hold. Therefore, u must have a neighbor in $\Lambda_{<j} \cup \Lambda_{j,<L_j}$. □

Claim 3.29 *Throughout the algorithm, for all* $1 \leq \ell < L_j$*, every vertex* $u \in U_{j,\ell}$ *has a path* P_u *of length at most* $O(\log n)$ *connecting it to a vertex in* $\Lambda_{<j} \cup \Lambda_{j,<\ell} \cup \hat{K}_{j,\ell}$*, such that every inner vertex on the path belongs to* $U_{j,\ell}$.

Proof: Fix an index $1 \leq \ell < L_j$, and consider the sublayer $\Lambda_{j,\ell}$ over the course of some (j,ℓ)-phase. Below, we add a superscript (0) to an object to denote that object at the beginning of the phase. Recall that the algorithm for computing a core decomposition from Theorem 3.13 ensured then there is an orientation of the edges of the graph $G[U_{j,\ell}]$, such that the resulting directed graph $\mathcal{D}_{j,\ell}$ is a DAG, and, for every vertex $u \in U_{j,\ell}$, $\text{in-deg}_{\mathcal{D}_{j,\ell}}(u) \leq \deg^{(0)}_{\leq (j,\ell)}(u)/12$. We denote by $\mathcal{D}^{(0)}_{j,\ell}$ the graph $\mathcal{D}_{j,\ell}$ at the beginning of the phase.

Let $\overline{\mathcal{D}}^{(0)}_{j,\ell}$ be the directed graph obtained from $\mathcal{D}^{(0)}_{j,\ell}$, by adding the set $\Lambda^{(0)}_{<j} \cup \Lambda^{(0)}_{j,<\ell} \cup \hat{K}^{(0)}_{j,\ell}$ of vertices to it, and all edges present in graph G at the beginning of the current (j,ℓ)-phase, connecting the vertices of $U^{(0)}_{j,\ell}$ to the vertices of $\Lambda^{(0)}_{<j} \cup \Lambda^{(0)}_{j,<\ell} \cup \hat{K}^{(0)}_{j,\ell}$. We orient these edges away from the vertices of $U_{j,\ell}$. Therefore, for every vertex $u \in U_{j,\ell}$, $\text{in-deg}_{\overline{\mathcal{D}}^{(0)}_{j,\ell}}(u) = \text{in-deg}_{\mathcal{D}^{(0)}_{j,\ell}}(u)$ holds.

Let $\overline{\mathcal{D}}_{j,\ell}$ denote the graph $\overline{\mathcal{D}}^{(0)}_{j,\ell}$ at some time during the execution of the (j,ℓ)-phase. Whenever an edge incident to a vertex of $\Lambda_{j,\ell}$ is deleted by the algorithm, we delete this edge from graph $\overline{\mathcal{D}}_{j,\ell}$ as well. Whenever a vertex of $\Lambda_{j,\ell}$ is removed from this set, we delete such a vertex and all its incident edges from $\overline{\mathcal{D}}_{j,\ell}$. From the definition of $\overline{\mathcal{D}}_{j,\ell}$, for every vertex $u \in U_{j,\ell}$, $\text{in-deg}_{\overline{\mathcal{D}}_{j,\ell}}(u) + \text{out-deg}_{\overline{\mathcal{D}}_{j,\ell}}(u) = \deg_{\leq (j,\ell)}(u)$ holds at all times.

Observe that, for every $u \in U_{j,\ell}$:

$$\begin{aligned}
\text{in-deg}_{\overline{\mathcal{D}}_{j,\ell}}(u) &\leq \text{in-deg}_{\overline{\mathcal{D}}^{(0)}_{j,\ell}}(u) && \text{as } \overline{\mathcal{D}}_{j,\ell} \subseteq \overline{\mathcal{D}}^{(0)}_{j,\ell} \\
&\leq \deg^{(0)}_{\leq (j,\ell)}(u)/12 && \text{by the property of } \overline{\mathcal{D}}^{(0)}_{j,\ell} \\
&\leq \deg_{\leq (j,\ell)}(u)/3 && \text{by Invariant I3,}
\end{aligned}$$

and so

$$\text{out-deg}_{\overline{\mathcal{D}}_{j,\ell}}(u) = \deg_{\leq (j,\ell)}(u) - \text{in-deg}_{\overline{\mathcal{D}}_{j,\ell}}(u) \geq 2 \cdot \text{in-deg}_{\overline{\mathcal{D}}_{j,\ell}}(u). \tag{3}$$

For any vertex set $S \subseteq V(\overline{\mathcal{D}}_{j,\ell})$, let $\text{in-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) = \sum_{u \in S} \text{in-deg}_{\overline{\mathcal{D}}_{j,\ell}}(u)$, $\text{out-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) = \sum_{u \in S} \text{out-deg}_{\overline{\mathcal{D}}_{j,\ell}}(u)$, and $\text{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) = \text{in-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) + \text{out-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S)$. For a vertex set $S \subseteq V(\overline{\mathcal{D}}_{j,\ell})$, we denote by S' the set of vertices containing all vertices of S, and all vertices $v \in V(\overline{\mathcal{D}}_{j,\ell})$, such that edge (u,v) with $u \in S$ belongs to the graph $\overline{\mathcal{D}}_{j,\ell}$. In other words, S' is an "out-ball" around S of radius 1.

Next, we show that, for any vertex set $S \subseteq U_{j,\ell}$, $\text{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S') \geq \frac{4}{3}\text{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S)$. Indeed:

$$\begin{aligned}\mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S') &= \mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) + \mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S' \setminus S)\\ &\geq \mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) + |E_{\overline{\mathcal{D}}_{j,\ell}}(S, S' \setminus S)|\\ &= \mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) + |E_{\overline{\mathcal{D}}_{j,\ell}}(S, S')| - |E_{\overline{\mathcal{D}}_{j,\ell}}(S, S)|\\ &\geq \mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) + \text{out-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) - \text{in-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S),\end{aligned}$$

where the last inequality follows from the fact that $|E_{\overline{\mathcal{D}}_{j,\ell}}(S, S')| = \text{out-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S)$ and $|E_{\overline{\mathcal{D}}_{j,\ell}}(S, S)| \leq \text{in-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S)$. From Equation (3), $\text{out-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) \geq 2\text{in-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S)$. Therefore, $\text{out-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) - \text{in-vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) \geq \mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S)/3$. We conclude that:

$$\mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S') \geq \mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S) + \mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S)/3 = \frac{4}{3}\mathrm{vol}_{\overline{\mathcal{D}}_{j,\ell}}(S).$$

It is now easy to see that, for any vertex $u \in U_{j,\ell}$, the distance from u to $\Lambda_{<j} \cup \Lambda_{j,<\ell} \cup \hat{K}_{j,\ell}$ in $\overline{\mathcal{D}}_{j,\ell}$ is bounded by $O(\log n)$. Otherwise, we can grow a ball from u, of volume $(4/3)^{\Omega(\log n)} \geq \mathrm{poly}(n)$, leading to a contradiction. □

We are now ready to complete the proof of Lemma 3.27. Suppose we are given a vertex $u \in V(G)$, and assume that it lies in some sublayer $\Lambda_{j,\ell}$, for $1 \leq j \leq r$ and $1 \leq \ell \leq L_j$. We start with $v_1 = u$, and then repeatedly apply Claim 3.28 and Claim 3.29 to the current vertex v_i, until we reach a vertex that lies in some core $\hat{K}$. For each application of the lemmas, starting from some vertex v_i, we obtain a path connecting v_i to either a vertex that lies in some core, or a vertex that belongs to some sublayer lying above the sublayer of v_i. In either case, the inner vertices of the path are contained in the sublayer of v_i. The final path P_u is obtained by concatenating all resulting paths. As there are $O(\log^2 n)$ sublayers and each path that we compute has length at most $O(\log n)$, the length of P_u is bounded by $O(\log^3 n)$. Moreover, it is easy to verify that the path visits the sublayers in a non-decreasing order. □

3.10 The Incident-Edge Data Structures

In order to efficiently construct and maintain the graphs $H_{j,\ell} = G[\Lambda_{j,\ell}]$, we maintain the following sets of edges:

- for every vertex $u \in V$ and index $1 \leq j \leq r$, edge set $\texttt{Edges}_j(u) = E_G(u, \Lambda_j)$.
- for every pair of indices $1 \leq j \leq r$, $1 \leq \ell \leq L_j$, for every vertex $u \in \Lambda_{j,\ell}$, edge set $\texttt{Edges}_{j,>\ell}(u) = E_G(u, \Lambda_{j,>\ell})$, and, for each $1 \leq \ell' \leq \ell$, edge set $\texttt{Edges}_{j,\ell'}(u) = E_G(u, \Lambda_{j,\ell'})$.

Intuitively, the data structures are defined in this way because we cannot afford to maintain the edge set $E_G(u, \Lambda_{j,\ell'})$ for every vertex $u \in \Lambda_{j,\ell}$, for all $1 \leq \ell < \ell'$, explicitly. This is because the vertices of $\Lambda_{j,>\ell}$ may move between the sublayers that lie below $\Lambda_{j,\ell}$ too frequently.

Throughout, the notation $\texttt{Edges}(\cdot)$ is used for the sets of edges that are explicitly maintained by the data structure, which we distinguish from subsets of edges of G, for which notation $E_G(\cdot)$ is used.

Consider some vertex $u \in V(G)$, and let $\Lambda_{j,\ell}$ be the sublayer containing u. We refer to the edge sets

$$\{\texttt{Edges}_{j'}(u)\}_{1 \leq j' \leq r}, \{\texttt{Edges}_{j,\ell'}(u)\}_{\ell' \leq \ell}, \texttt{Edges}_{j,>\ell}(u)$$

as the *incident-edge data structure* of vertex u. Below, we show that the incident-edge data structures for all vertices of G can be maintained in total update time $\tilde{O}(m\Delta^2)$ over the course of the algorithm's execution.

Recall that, from Observation 3.5, the layers $\Lambda_1, \ldots, \Lambda_r$, and the edge sets $\texttt{Edges}_j(u)$ for all vertices $u \in V(G)$ and layers $1 \leq j \leq r$ can be maintained over the course of the algorithm, in time $\tilde{O}(m+n)$.

Next, we fix an index $1 \leq j \leq r$, and show how to maintain the edge sets $\{\texttt{Edges}_{j,\ell'}(u)\}_{\ell' \leq \ell}, \texttt{Edges}_{j,>\ell}(u)$ for all $1 \leq \ell \leq L_j$ and $u \in \Lambda_{j,\ell}$.

At the beginning of the algorithm, all vertices of Λ_j lie in the sublayer $\Lambda_{j,1}$. For every vertex $u \in \Lambda_{j,1}$, set $\texttt{Edges}_{j,>1}(u) = \emptyset$, and edge set $\texttt{Edges}_{j,1}(u)$ contains all edges in $E(H_{j,1})$ that are incident to u. All these edge sets can be initialized in time $O(|E(H_{j,1})|)$

There are two cases when we need to update these sets: (1) when a vertex is moved to Λ_j^- and (2) when we set $\Lambda_{j,\ell} \leftarrow \Lambda_{j,\geq \ell}$ and initialize the sublayer $\Lambda_{j,\ell}$ for some ℓ.

For the first event, consider some sublayer $\Lambda_{j,\ell}$, and a vertex $u \in \Lambda_{j,\ell}$ that is moved to the buffer sublayer Λ_j^-. We need to partition the edges from the original edge set $\texttt{Edges}_{j,>\ell}(u) = E_G(u, \Lambda_{j,>\ell})$ into edge sets $\texttt{Edges}_{j,\ell+1}(u), \ldots, \texttt{Edges}_{j,L_j}(u)$, where $\texttt{Edges}_{j,\ell'}(u) \leftarrow E_G(u, \Lambda_{j,\ell'})$ for each $\ell' > \ell$. Also, for each vertex $v \in \{w \mid (u,w) \in E_G(u, \Lambda_{j,>\ell})\}$, we need to move the edge (u,v) from $\texttt{Edges}_{j,\ell}(v)$ to $\texttt{Edges}_{j,>\ell'}(v)$, where ℓ' is the index of the sublayer $\Lambda_{j,\ell'}$ containing v; if $v \in \Lambda_{j,L_j}$, then we instead move (u,v) from $\texttt{Edges}_{j,\ell}(v)$ to $\texttt{Edges}_{j,L_j}(v)$. All these operations can be done in time $|E_G(u, \Lambda_{j,>\ell})|$.

Recall that every time we move u from some sublayer $\Lambda_{j,\ell}$ to the buffer sublayer Λ_j^-, we remove $|E_G(u, \Lambda_{j,>\ell})|$ edges from the edge set $\Pi_{j,\ell}$ that we defined in Section 3.7. Therefore, we can charge the total time needed to update the incident-edge data structures due to moves of vertices into Λ_j^- to the set $\bigcup_{j=1}^{L_j} \hat{\Pi}_{j,\ell}$ of edges. From Corollary 3.25, the total number of edges in this set is $\texttt{inc}_j \leq O(n_{\leq j} h_j \Delta/\varphi^3)$. Therefore, the total time spent on updating the incident-edge data structures due to moves of vertices into to Λ_j^- is at most $O(n_{\leq j} h_j \Delta/\varphi^3)$. It is also possible that a vertex $u \in \Lambda_{<j}$ is moved to Λ_j^-. However, this only happen once per vertex, so the total update time due to such moves is $O(m)$.

For the second event, fix some index $1 \leq \ell < L_j$, and consider the time when a new (j,ℓ)-phase starts. Recall that we set $\Lambda_{j,\ell} \leftarrow \Lambda_{j,\geq \ell}$. For every vertex u that originally lied in $\Lambda_{j,\geq \ell}$, we need to set $\texttt{Edges}_{j,\ell}(u) \leftarrow \texttt{Edges}_{j,\ell}(u) \cup \texttt{Edges}_{j,>\ell}(u)$. This can be done, for all such vertices u, in total time $O(|E_G(\Lambda_{j,\geq \ell})|)$. Recall that, from Invariant I1, $|\Lambda_{j,\geq \ell}| \leq O(n_{\leq j}/2^\ell)$, and that, from Observation 3.3, edge set $E_G(\Lambda_j)$ has an $(h_j\Delta)$-orientation. It is then easy to see that edge set $E_G(\Lambda_{j,\geq \ell})$ has an $(h_j \cdot \Delta)$-orientation as well, and so $|E_G(\Lambda_{j,\geq \ell})| \leq O(n_{\leq j} h_j \Delta/2^\ell)$. By Lemma 3.18, there are at most $\widehat{O}(2^\ell \Delta)$ (j,ℓ)-phases over the course of the entire algorithm, and so the total time that we need to spend on updating the incident-edge data structure due to the initialization of the sublayer $\Lambda_{j,\ell}$ is bounded by $\widehat{O}(n_{\leq j} h_j \Delta^2)$. Summing over all sublayers of layer Λ_j, the total time that the algorithm spends on maintaining the edge-incident data structures for vertices lying in layer Λ_j is bounded by $O(n_{\leq j} h_j \Delta^2 \log(n)/\varphi^3 + m)$.

Lastly, observe that for all $1 \leq j \leq r$, $n_{\leq j} h_j \leq O(m)$. Therefore, the total time that is needed to maintain the incident-edge data structure for all vertices og G is at most:

$$\sum_j \Big(O(n_{\leq j} h_j \Delta/\varphi) + \widehat{O}(n_{\leq j} h_j \Delta^2 + m) \Big) = \widehat{O}(m\Delta^2).$$

3.11 Total Update Time, and Data Structures to Support Short-Core-Path and To-Core-Path Queries

In this subsection, we provide some additional data structures that are needed to support Short-Core-Path and To-Core-Path Queries, and analyze the total update time of the main algorithm for Theorem 3.4. We start by analyzing the total update time required for maintaining all data structures that we have described so far. Then, we describe additional data structures that we maintain for supporting Short-Core-Path, To-Core-Path, and Short-Path queries, and analyze their update time.

Maintaining the Sublayers. Recall that, from Observation 3.5, the total update time that is needed to maintain the partition of $V(G)$ into $\Lambda_1, \ldots, \Lambda_r$ is $\tilde{O}(m)$. As shown in Section 3.10,the edge-incident data structures for all vertices require total update time $\widehat{O}(m\Delta^2)$.

Consider now some index $1 \leq j \leq r$. For the buffer sublayer Λ_j^-, we do not need to maintain any additional data structures. Consider now some non-buffer sublayer $\Lambda_{j,\ell}$, for $\ell < L_j$. At the beginning of a (j,ℓ)-phase, we construct the graph $H_{j,\ell}$ by setting $E(H_{j,\ell}) \leftarrow \bigcup_{u \in \Lambda_{j,\ell}} \mathtt{Edges}_{j,\ell}(u)$, using the incident-edge data structure. This takes $O(|E(H_{j,\ell})|)$ time. Note that, without the incident-edge data structure, it is not immediately clear how to construct the graph $H_{j,\ell}$ in this time. The resulting graph $H_{j,\ell}$ is precisely $G[\Lambda_{j,\ell}]$, as desired. Next, we perform the core decomposition in graph $H_{j,\ell}$ using the algorithm from Theorem 3.13. The running time of the algorithm is $\widehat{O}(|E(H_{j,\ell})|)$. Recall that, from Observation 3.3, the edge set $E_G(\Lambda_j)$ has an $(h_j\Delta)$-orientation. Moreover, from Invariant I1, $|\Lambda_{j,\ell}| \leq \frac{n_{\leq j}}{2^{\ell-1}}$. Therefore, $|E(H_{j,\ell})| \leq |\Lambda_{j,\ell}| \cdot h_j\Delta \leq \frac{n_{\leq j}}{2^{\ell-1}} \cdot h_j\Delta$. By Lemma 3.18, the total number of (j,ℓ)-phases over the course of the algorithm is bounded by $\widehat{O}(2^\ell\Delta)$. Therefore, the total time that is needed to construct the graphs $H_{j,\ell}$ and to compute core decompositions of such graphs over the course of the entire algorithm is bounded by $\widehat{O}(n_{\leq \ell}h_j\Delta/2^\ell) \cdot \widehat{O}(2^\ell\Delta) = \widehat{O}(n_{\leq j}h_j\Delta^2) \leq \widehat{O}(m\Delta^2)$.

Note that it is straightforward to check that invariants I1 and I3 hold over the course of the algorithm: For each $1 \leq j \leq r$ and $1 \leq \ell \leq L_j$ we need to ensure that $|\Lambda_{j,\ell}| \leq n_{\leq j}/2^{\ell-1}$ always holds. This can be checked in constant time by keeping track of $|\Lambda_{j,\ell}|$. For each vertex $u \in U_{j,\ell}$, we need to ensure the invariant that $\deg_{\leq (j,\ell)}(u) \geq \deg^{(0)}_{\leq (j,\ell)}(u)/4$ always holds. This can be checked in $O(\log n)$ by maintaining prefix sums of $|\mathtt{Edges}_{j'}(u)|$ and $|\mathtt{Edges}_{j,\ell'}(u)|$ for all $j' < j$ and $\ell' \leq \ell$. As there are $\widehat{O}(\log^2 n)$ sublayers, the total cost for maintaining all sublayers $\Lambda_{j,\ell}$ and their corresponding graphs $H_{j,\ell} = G[\Lambda_{j,\ell}]$, together with computing the initial core decompositions $\mathcal{F}_{j,\ell}$ is at most $\widehat{O}(m\Delta^2)$.

Oracles for Short-Core-Path queries. Whenever a core K is created, we maintain the data structure from Theorem 3.9 that allows us to maintain the core K under the deletion of edges from G, and to support Short-Core-Path(K,u,v) queries within the core K. Consider some index $1 \leq j \leq r$ and a non-buffer sublayer $\Lambda_{j,\ell}$ of Λ_j. At the beginning of each (j,ℓ)-phase, let $\mathcal{F}_{j,\ell}$ denote the collection of cores constructed by the algorithm from Theorem 3.13. The total update time needed to maintain the data structure for all cores $K \in \mathcal{F}_{j,\ell}$ throughout a single (j,ℓ)-phase is $\sum_{K \in \mathcal{F}_{j,\ell}} O(|E(K)|^{1+1/q}(\gamma(n))^{O(q)}) \leq O(|E(H_{j,\ell})|^{1+1/q}(\gamma(n))^{O(q)})$. As observed already, $|E(H_{j,\ell})| \leq |\Lambda_{j,\ell}| \cdot h_j\Delta \leq n_{\leq j}h_j\Delta/2^{\ell-1} \leq O(m\Delta/2^\ell)$ Since, from Lemma 3.18, the total number of (j,ℓ)-phases over the course of the algorithm is bounded by $\widehat{O}(2^\ell\Delta)$, the total time needed to maintain all cores within the layer (j,ℓ) over the course of the algorithm is bounded by $O(m^{1+1/q}\Delta^{2+1/q}(\gamma(n))^{O(q)})$. Summing this up over all $O(\log n)$ non-buffer sublayers $\Lambda_{j,\ell}$, we get that the total time that is needed to maintain all cores that are ever present in $\mathcal{F}_j$ is bounded by $O(m^{1+1/q}\Delta^{2+1/q}(\gamma(n))^{O(q)})$.

Note that the algorithm from Theorem 3.9 directly supports the Short-Core-Path(K,u,v) queries: given any pair of vertices u and v that lie in the same core K, it return a simple u-v path P in K connecting

u to v, of length at most $(\gamma(n))^{O(q)}$, in time $(\gamma(n))^{O(q)}$.

ES-trees for To-Core-Path queries. At the beginning of a (j,ℓ)-phase of a non-buffer sublayer $\Lambda_{j,\ell}$, we construct an auxiliary graph $C_{j,\ell}$ for maintaining short paths from vertices of $U_{j,\ell}$ to vertices of $\Lambda_{<j} \cup \Lambda_{j,<\ell} \cup \hat{K}_{j,\ell}$, whose existence is guaranteed by Claim 3.29. The vertex set of the graph $C_{j,\ell}$ is $V(C_{j,\ell}) = \Lambda_{j,\ell} \cup \{s_{j,\ell}\}$. The edge set of $C_{j,\ell}$ contains the edges of $E(H_{j,\ell})$. Additionally, for every vertex $u \in \hat{K}_{j,\ell}$, we add the edge $(s_{j,\ell}, u)$ into $C_{j,\ell}$. For every vertex $u \in U_{j,\ell}$, if $E_G(u, \Lambda_{<j} \cup \Lambda_{j,<\ell}) \neq \emptyset$, then we add the edge $(s_{j,\ell}, u)$ to $C_{j,\ell}$ and we associate the edge $(s_{j,\ell}, u)$ with an edge (w, u) where w is some neighbor of u in $\Lambda_{<j} \cup \Lambda_{j,<\ell}$. Note that we can maintain this association by using the incident-edge data structure. It is easy to see that the time needed to construct the graph $C_{j,\ell}$ is subsumed by the time needed to construct the graph $H_{j,\ell}$.

Observe that, throughout each (j,ℓ)-phase, graph $C_{j,\ell}$ only undergoes edge- and vertex-deletions, just like graph $H_{j,\ell}$. Therefore, we can maintain an ES-tree $T_{j,\ell}$ rooted at $s_{j,\ell}$, in the graph $C_{j,\ell}$, up to distance $O(\log n)$. The total update time for $T_{j,\ell}$ is $O(|E(C_{j,\ell})| \log n)$ for each (j,ℓ)-phase. As $|E(C_{j,\ell})| \leq |E(H_{j,\ell})| + |U_{j,\ell}| = O(n_{\leq j} h_j \Delta / 2^\ell)$ and there are at most $\widehat{O}(2^\ell \Delta)$ (j,ℓ)-phases, the total time needed to maintain the ES-trees $T_{j,\ell}$ in graphs $C_{j,\ell}$ over the course of the algorithm is at most $\widehat{O}(n_{\leq j} h_j \Delta^2) = \widehat{O}(m\Delta^2)$. The total cost of maintaining such trees for all non-buffer sublayers of all layers Λ_j is at most $\widehat{O}(m\Delta^2)$.

Supporting To-Core-Path queries. For convenience, we will slightly abuse notation. For each non-buffer sublayer $\Lambda_{j,\ell}$, the ES-tree $T_{j,\ell}$ is formally a subgraph of $C_{j,\ell}$, but we will treat $T_{j,\ell}$ as a subgraph of $G[\Lambda_{\leq j}]$ as follows. Each edge $(s_{j,\ell}, u) \in T_{j,\ell}$ where $u \in U_{j,\ell}$ corresponds to some edge $(u, w) \in E(U_{j,\ell}, \Lambda_{<j} \cup \Lambda_{j,<\ell}) \subseteq E(G[\Lambda_{\leq j}])$. Edges of the form $(s_{j,\ell}, u) \in T_{j,\ell}$, where $u \in \hat{K}_{j,\ell}$ do not correspond to edges of $G[\Lambda_{\leq j}]$. The remaining edges of $T_{j,\ell}$ that are not incident to $s_{j,\ell}$ are edges of $G[\Lambda_{\leq j}]$ by definition. For each buffer sublayer $\Lambda_j^- = \Lambda_{j,L_j}$, we will also define a subgraph $T_{j,\ell}$ as follows. For each $u \in \Lambda_{j,L_{j'}}$, $T_{j,\ell}$ contains the edge (u, v) where v is the (lexicographically) first neighbor of u in $\Lambda_{<j} \cup \Lambda_{j,<L_j}$ (v must exist by Claim 3.28). Note that $T_{j,\ell}$ is a subgraph of $G[\Lambda_{\leq j}]$.

Now, in order to respond to To-Core-Path(u) query, where $u \in \Lambda_{j,\ell}$, we follow the simple path of length $O(\log n)$ in $T_{j,\ell}$ starting from u to a vertex of $\hat{K}_{j,\ell}$, or a vertex of $\Lambda_{<j} \cup \Lambda_{j,<\ell}$. If we reach a vertex of $\hat{K}_{j,\ell}$, then we are done. Otherwise, we reach a vertex in some sublayer that lies above $\Lambda_{j,\ell}$, and we continue the same process in that sublayer. The total number of such iterations is then bounded by the total number of sublayers in the entire graph – at most $O(\log^2 n)$. Observe that the paths computed at different iterations may only intersect at their endpoints, because every vertex on such a path, except for possibly the last vertex, lies in a single sublayer. Therefore, the concatenation of these paths is a simple path. To conclude, given a query To-Core-Path(u), we can return a simple path P of length $O(\log^3 n)$ connecting u to a vertex in some core, in time $O(|P|)$, such that P is non-increasing with respect to the sublayers. In other words, if $P_u = \{u = u_1, u_2, \ldots, u_k\}$, then, for each i, if $u_i \in \Lambda_{j,\ell}$, then $u_{i+1} \in \Lambda_{<j} \cup \Lambda_{j,\leq \ell}$.

Minimum Spanning Forest for Short-Path queries. For an index $1 \leq j \leq r$, we denote $T_{\leq j} = \bigcup_{j' \leq j, \ell \geq 1} T_{j',\ell}$. Recall that $\mathcal{F}_{\leq j}$ is the collection of all cores for layers $\Lambda_1, \ldots, \Lambda_j$. Let $\hat{K}_{\leq j} = \bigcup_{K \in \mathcal{F}_{\leq j}} K$ denote the union of all the cores in $\mathcal{F}_{\leq j}$. Note that $\hat{K}_{\leq j}$ and $T_{\leq j}$ subgraphs of $G[\Lambda_{\leq j}]$, and they do not share any edges.

We maintain a fully dynamic minimum spanning forest M_j for the graph $G[\Lambda_{\leq j}]$, with the following edge lengths. We assign weight 0 to all edges in $\hat{K}_{\leq j}$, weight 1 to all edges of $T_{\leq j}$, and weight 2

to all remaining edges of $G[\Lambda_{\leq j}]$. The spanning forest M_j can be maintained using the algorithm of [HdLT01], that has $O(\log^4 n)$ amortized update time.

Additionally, we use the *top tree* data structure due to [AHdLT05], whose properties are summarized in the following lemma.

Lemma 3.30 (Top Tree Data Structure from [AHdLT05]) *The top tree data structure $\mathcal{T}$ is given as input a forest F with weights on edges, that undergoes edge insertions and edge deletions (but we are guaranteed that F remains a forest throughout the algorithm), and supports the following queries, in $O(\log n)$ time per query:*

- $\texttt{connect}(x,y)$*: given two vertices x and y, determine whether x and y are in the same connected component of F (see Section 2.4 of [AHdLT05]);*
- $\texttt{minedge}(x,y)$*: given two vertices x and y lying in the same connected component of F, return a minimum-weight edge on the unique path connecting x to y in F (a variation of Theorem 4 of [AHdLT05]);*
- $\texttt{weight}(x,y)$*: given two vertices x and y lying in the same connected component of F, return the total weight of all edges lying on the unique path connecting x to y in F (Lemma 5 of [AHdLT05]).*
- $\texttt{jump}(x,y,d)$*: given two vertices x and y lying in the same connected component of F, return the dth vertex on the unique path connecting x to y in F; if the path connecting x to y contains fewr than d vertices, return $\emptyset$ (Theorem 15 of [AHdLT05]).*

The data structure has $O(\log n)$ worst-case update time.

For all $1 \leq j \leq r$, we maintain the top tree data structure $\mathcal{T}_{M_j}$ for the forest M_j.

It is easy to see that the total time that is required for maintaining the minimum spanning forests $\{M_j\}_{j=1}^{r}$ and their corresponding top tree data structures $\mathcal{T}_{M_j}$ is subsumed by other components of the algorithm.

To conclude, the total update time of the LCD data structure for Theorem 3.4 is $\widehat{O}(m^{1+1/q}\Delta^{2+1/q}(\gamma(n))^{O(q)})$.

3.12 Supporting Short-Path Queries

In this section, we fix an index $1 \leq j \leq r$, and describe an algorithm for processing Short-Path$(j,\cdot,\cdot)$ queries. Recall that we have denoted $\hat{K}_{\leq j} = \bigcup_{K \in \mathcal{F}_{\leq j}} K$ and $T_{\leq j} = \bigcup_{j' \leq j, \ell \geq 1} T_{j',\ell}$. We start by analyzing the structure of the spanning forest M_j.

Recall that $T_{\leq j}$ is a forest, and every tree in this forest is rooted in a vertex of $\hat{K}_{\leq j}$. Moreover, if a vertex of $T_{\leq j}$ does not serve as a tree root, then it does not lie in $\hat{K}_{\leq j}$, and every vertex in $\Lambda_{\leq j} \setminus \hat{K}_{\leq j}$ must lie in some tree in $T_{\leq j}$. Recall also that, from Lemma 3.27, the depth of every tree in $T_{\leq j}$ is bounded by $O(\log^3 n)$.

Consider now some connected component C of graph $G[\Lambda_{\leq j}]$. Let $\mathcal{F}^C$ denote the collection of all cores $K \in \mathcal{F}_{\leq j}$ with $K \subseteq C$, and let $k_C = |\mathcal{F}^C|$. Recall that, from Observation 3.16, $k_C \leq O(|V(C)|/(\varphi^2 h_j)) = O(|V(C)|(\gamma(n))^2/h_j)$.

The following two observations easily follow from the properties of a minimum spanning tree.

Observation 3.31 *Let C be a connected component of $G[\Lambda_{\leq j}]$, and let $K \in \mathcal{F}^C$ be a core that currently lies in $\mathcal{F}_{\leq j}$ and is contained in C. Then there is some connected sub-tree T^* of the forest M_j that contains every vertex of K.*

Proof: Assume otherwise. Consider the sub-graph of M_j induced by the edges of $E(K)$. Then this graph is not connected, and moreover, there must be two connected components X and Y of this graph, such that core K contains some edge e connecting a vertex of X to a vertex of Y. Adding the edge e to M_j must create some cycle R. We claim that at least one edge on this cycle must have weight greater than 0. Indeed, otherwise, every edge on cycle R lies in the core K, and so X and Y cannot be two connected components of the subgraph of M_j induced by $E(K)$. Since edge e has weight 0, we have reached a contradiction to the minimality of the forest M_j. □

Observation 3.32 *Every edge of the forest $T_{\leq j}$ belongs to M_j.*

Proof: Assume for contradiction that this is not the case, and let T' be a tree of $T_{\leq j}$ with $E(T') \not\subseteq E(M_j)$ As before, consider the sub-graph of M_j induced by the edges of $E(T')$. This graph is not connected, and, so there must be two connected components X and Y of this graph, such that tree T' contains some edge e connecting a vertex of X to a vertex of Y. Adding the edge e to M_j must create some cycle R. We claim that at least one edge on this cycle must have weight 2. Indeed, otherwise, every edge on cycle R has weight 0 or 1. This is impossible because tree T' contains exactly one vertex that lies in a core of $\mathcal{F}_{\leq j}$, and the only edges whose weight is 0 are edges that are contained in the cores. Therefore, there must be some edge e' on the cycle R that is incident to some vertex of T', is not contained in T', and is not contained in any core of $\mathcal{F}_{\leq j}$. The weight of such an edge then must be 2. But, since the weight of the edge e is 1, this contradicts the minimality of M_j. □

Consider again some connected component C of the graph $G[\Lambda_{\leq j}]$, and recall that M_j is a minimum spanning forest of $G[\Lambda_{\leq j}]$. Let $M_j^C \subseteq M_j$ be the unique tree in the forest M_j that is spanning C. From the above two observations it is easy to see that, if we delete all weight-2 edges from M_j^C, then we will obtain k_C connected components. Therefore, we obtain the following immediate corollary.

Corollary 3.33 *For every connected component C of $G[\Lambda_{\leq j}]$, tree M_j^C contains at most $k_C - 1$ edges of weight 2.*

Consider now any path P in the forest M_j. Recall that all edges of P have weights in $\{0, 1, 2\}$. For $x \in \{0, 1, 2\}$, an *x-block* of the path P is a maximal subpath of P such that every edge on the subpath has weight exactly x. We need the following observation on the structure of such paths.

Observation 3.34 *Let P be any path in the spanning forest M_j, and let C be the connected component of $G[\Lambda_{\leq j}]$ containing P. Then:*

1. *there are at most $k_C - 1$ edges of weight 2 in P;*
2. *the number of 0-blocks in P is at most k_C; and*
3. *the number of 1-blocks in P is at most $2k_C$, with each 1-block having length at most $O(\log^3 n)$.*

Proof: The first assertion follows immediately from Corollary 3.33, and the second assertion follows immediately from Observation 3.31 and the fact that at most k_C cores of $\mathcal{F}_{\leq j}$ are contained in C.

In order to prove the third assertion, let Σ be a collection of paths that is obtained by removing all weight-2 edges from P. Then, from Corollary 3.33, $|\Sigma| \leq k_C - 1$. Moreover, since every tree in $T_{\leq j}$ contains exactly one vertex of $\hat{K}_{\leq j}$, for each such path $\sigma \in \Sigma$, there is at most one core $K \in \mathcal{F}_{\leq j}$, such that the edges of K lie on σ, and if such a core exists, then, from Observation 3.31, the edges of K appear contiguously on σ. Therefore, every path σ contains at most two 1-blocks, and the total number of 1-blocks on P is at most $2k_C$. Since every tree in $T_{\leq j}$ has depth at most $O(\log^3 n)$, the length of each such 1-block is at most $O(\log^3 n)$. □

The following corollary follows immediately from Observation 3.34 and the fact that for every connected component C of $\Lambda_{\leq j}$, $k_C = O(|V(C)|(\gamma(n))^2/h_j)$.

Corollary 3.35 *Let P be any path contained in the graph M_j, and let C be the connected component of $\Lambda_{\leq j}$ containing P. Then the total number of edges of P that have non-zero weight is at most $\tilde{O}(k_C) \leq \tilde{O}(|V(C)|(\gamma(n))^2/h_j)$.*

We are now ready to describe an algorithm for processing a query $\mathsf{Short\text{-}Path}(j, u, v)$. Our first step is to check whether u and v are connected in M_j. This can be done in time $O(\log n)$, using the $\texttt{connect}(u, v)$ query in the top tree $\mathcal{T}_{M_j}$ data structure. If u and v are not connected in M_j, then we terminate the algorithm and report that u and v are not connected in $G[\Lambda_{\leq j}]$. We assume from now on that u and v are connected in M_j.

We denote by P the unique path connecting u and v in M_j. Note that our algorithm does not compute the path P explicitly, as it may be too long. We think of the path P as being oriented from u to v. Let $B_1, \ldots, B_z$ be the sequence of all maximal 0-blocks on path P; we assume that the blocks are indexed in the order of their appearance on P. For $1 \leq i \leq z$, we denote by b_i and by b'_i the first and the last endpoint of B_i, respectively. For $1 \leq i < z$, let A_i be the sub-path of P connecting b'_i to b_{i+1}; let A_0 be the sub-path of P connecting u to b_1, and let A_z be the sub-path of P connecting b'_z to v. The next step in our algorithm is to identify all endpoints of the 0-blocks on path P, that is, the algorithm will find the parameter z (the number of the maximal 0-blocks on P), and, for all $1 \leq i \leq z$, it will compute the endpoints b_i, b'_i of block B. We do so using queries $\texttt{minegdge}$, $\texttt{weight}$, and $\texttt{jump}$ to the top tree $\mathcal{T}_{M_j}$ data structure.

Specifically, we start by running query $\texttt{minedge}(u, v)$ on the top tree $\mathcal{T}_{M_j}$. Let $e = (x, y)$ be the returned edge. If the weight of the edge is greater than 0, then there are no 0-weight edges on path P, and so we can skip the current step. Assume therefore that the weight of the edge e is 0. Let B_i be the 0-block containing e (note that we do not know the index i). In order to find the first endpoint b_i of the 0-block B_i, we perform a binary search using queries $\texttt{jump}(x, u, d)$ for various values of d. If a_d is the vertex returned in response to query $\texttt{jump}(x, u, d)$, then we use query $\texttt{weight}(x, a_d)$ to find the total weight of all edges on the sub-path of P connecting x to a_d. If the returned weight is 0, then we know that $a_d \in B_i$, and we increase the value of d; otherwise, we know that the sub-path of P between b_i and x contains fewer than d edges, and we decrease d accordingly. Therefore, using binary search, in $O(\log n)$ iterations, we can compute the endpoint b_i of the block B_i, and we can compute the other endpoint b'_i of the block similarly. The total time needed to compute both endpoints is therefore $O(\log^2 n)$. Once we compute the endpoints b_i, b'_i, we recursively apply the same algorithm to the sub-path of P connecting u to b_i, and the sub-path of P connecting b'_i to v. We conclude that we can compute the number z of the maximal 0-blocks on path P, and the endpoints of these blocks, in time $O(z \log^2 n)$.

Once the endpoints of all 0-blocks are computed, we compute the paths $A_1, \ldots, A_z$, using queries $\texttt{jump}(a, a', 1)$. Lastly, for all $1 \leq i \leq z$, we run query $\mathsf{Short\text{-}Core\text{-}Path}(K_i, b_i, b'_i)$, where K_i is the core of $\mathcal{F}_{\leq j}$ containing b_i and b'_i, to compute a path B'_i in core K_i that connects b_i to b'_i and has length at most $(\gamma(n))^{O(q)}$. We then return a u-v path that is obtained by concatenating the paths $A_1, B'_1, A_2, \ldots, B'_z, A_z$.

We use the following lemma to bound the length of the resulting path.

Lemma 3.36 *Given query $\mathsf{Short\text{-}Path}(j, u, v)$, the above algorithm either correctly reports that u and v are not connected in $G[\Lambda_{\leq j}]$ in time $O(\log n)$, or it returns a simple u-v path P' of length at most $O(|V(C)|(\gamma(n))^{O(q)}/h_j)$, in time $O(|P'| \cdot (\gamma(n))^{O(q)})$, where C is the connected component of $G[\Lambda_{\leq j}]$ containing u and v.*

Proof: It is immediate to see that, if u and v are not connected in $G[\Lambda_{\leq j}]$, then the algorithm reports this correctly in time $O(\log n)$. Therefore, we assume from now on that some connected component C of $G[\Lambda_{\leq j}]$ contains u and v. As before, we denote by P the unique u-v path in the graph M_j. Note that, from Corollary 3.35, the total number of edges on P with non-zero weight is at most $\tilde{O}(|V(C)|(\gamma(n))^2/h_j)$. In particular, the number of maximal 0-blocks on P must be bounded by $z \leq \tilde{O}(|V(C)|(\gamma(n))^2/h_j)$. Since we are guaranteed that, for all $1 \leq i \leq z$, the length of the path B'_i is bounded by $(\gamma(n))^{O(q)}$, we conclude that the length of the returned path P' is at most $O(|V(C)|(\gamma(n))^{O(q)}/h_j)$.

In order to bound the running time, recall that detecting the endpoints of the 0-blocks takes time $O(z \cdot \log^2 n)$. Computing all vertices on paths $A_1, \ldots, A_z$ takes time $O(\log n)$ per vertex. Lastly, computing the paths $B'_1, \ldots, B'_z$ takes total time at most $z \cdot (\gamma(n))^{O(q)}$. Altogether, the running time is bounded by $O(|P'| \cdot (\gamma(n))^{O(q)})$. □

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. We still include a proof for completeness, since our LCD data structure is defined somewhat differently. 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 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 3.4 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 $\mathsf{Short\text{-}Path}(j,u,v)$, 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 $\mathsf{Short\text{-}Path}$ 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 $\mathsf{Short\text{-}Path}(j_i,u,v)$ 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 $\mathsf{Short\text{-}Path}(C,u,v)$ 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 $\mathsf{CONN\text{-}SF}$ 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 4.2 *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 4.3 (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 4.3 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 $\mathsf{path\text{-}query}_D(s,v)$ and $\mathsf{dist\text{-}query}_D(s,v)$.

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 4.3. 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

[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 $\mathsf{Proc\text{-}Degree\text{-}Pruning}$ 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).

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 4.2, 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:

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

(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 super-nodes 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 explicity). As before if $\mathsf{dist}_G(s,v) \geq D$, then, from Observation 4.2, $\mathsf{dist}'(s,v) \leq (1+\epsilon/2)\mathsf{dist}_G(s,v)$.

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 3.4.

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 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 $1 \leq i \leq \lceil \log_{1+\epsilon} n \rceil$, let $D_i = (1+\epsilon)^i$. For all $1 \leq i \leq \lceil \log_{1+\epsilon} n \rceil$, 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 form 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 4.2, 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 $\mathsf{ES\text{-}Tree}$, 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 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 $\mathsf{ES\text{-}Trees}$ 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 5.3 *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 $|S| \leq O(n/R^c)$. $\square$

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 $\mathsf{conn}(G,x,y)$ 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 $T(s)$. 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 3.4. We invoke the algorithm from Theorem 3.4 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 3.4. At a high level, our algorithm can be described in one sentence:

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

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 $O(nh_j\Delta R_j^d) = O(n^{2+1/k}D(\gamma(n))^{O(k)}) \leq \widehat{O}(n^{2+1/k}D)$ (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 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 3.4, 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:

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

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 tiem 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 $\mathsf{dist}_{Z_j}(V(K), v) \leq R^c_j$ (i.e. $\mathsf{dist}_{Z^K_j}(s_K, v) \leq R^c_j + 1$). For each $v \in \Lambda_{\geq j}$, we maintain a list of all ES-Trees T_K that covers 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 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 index j^* for which $x \in \Lambda_{j^*}$; therefore, $V(P^*) \subseteq \Lambda_{\geq j^*}$. We start with the following crucial observation.

Lemma 5.4 *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 in 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 5.4, 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.

A Proofs Omitted from Section 2

A.1 Proof of Observation 2.3: Degree Pruning

It is immediate that the degree of every vertex in graph $H[A]$ is at least d. We now prove that A is the unique maximal set with this property at any time. Assume for contradiction that at some time there is a subset $A' \subseteq V(H)$ where every vertex in $H[A']$ has degree at least d but $A' \not\subseteq A$. Denote

$\{v_1, \ldots, v_r\} = V(H) \setminus A$ where the vertices are indexed in the order in which they were removed from A. Then there must be some vertex $v \in A' \setminus A$. Let v_i be such a vertex with the smallest index i. But then $v_1, \ldots, v_{i-1} \notin A'$, so v_i must have fewer than d neighbors in A', a contradiction.

B Proofs Omitted from Section 3

B.1 Proof of Observation 3.3: Bounding Number of Edges Incident to Layers

Fix some index $1 \leq j \leq r$. In order to define an $(h_j\Delta)$-orientation of $E_{\geq j}$, we first define an ordering ρ of the vertices of $V(G)$. Consider the following experiment. We run Alg-Maintain-Pruned-Set(G, h_{j-1}) in order to maintain the vertex set A_{j-1}, as G undergoes edge deletions. For a vertex $v \in V(G)$, we define its *drop time* to be the first time in the execution of this algorithm when v did not belong to set A_{j-1}; if no such time exists, then the drop time of v is infinite. Recall that, from Observation 2.4, if the drop time of v is finite and equal to t, then at time t, v had fewer than $h_{j-1} = \Delta h_j$ neighbors in A_{j-1}. We let ρ be the ordering of the vertices of $V(G)$ by their drop time, from smallest to largest, breaking ties arbitrarily. Notice that every edge in $E_{\geq j}$ must have an endpoint with a finite drop time. Consider now some edge $e = (u, v) \in E_{\geq j}$. If u appears before v in the ordering ρ, then we assign the direction of the edge e to be from u to v; note that, from the definition of $E_{\geq j}$, the drop time of u must be finite. This gives a $(h_j\Delta)$-orientation for $E_{\geq j}$. It now follows immediately that $|E_{\geq j}| \leq \Delta h_j n$.

Next, let S_j be the set of vertices that join the layer Λ_j at any time of the algorithm's execution. Observe that $|S_j| \leq n_{\leq j}$ must hold because virtual degrees may only decrease, and so $|E_{\geq j}(S_j)| \leq n_{\leq j} \cdot h_j\Delta$. As the edges whose both endpoints are contained in Λ_j at any point of time must belong to $E_{\geq j}(S_j)$, the number of such edges is at most $n_{\leq j}h_j\Delta$. We conclude that the number of edges e, such that, at any time during the algorithm's execution, both endpoints of e are contained in Λ_j is at most $n_{\leq j}h_j\Delta$.

B.2 Existence of Expanding Core Decomposition

The goal of this section is to prove the following theorem about the existence of a core decomposition in a high-degree graph. We note that a theorem that is very similar in spirit (but different in the exact definitions and parameters) was shown in [CK19], and the proof that we provide uses similar ideas.

Theorem B.1 (Expanding Core Decomposition) *Let H be an n-vertex simple graph with minimum degree at least h. There exists a collection $\mathcal{F} = \{K_1, \ldots, K_t\}$ of vertex-disjoint induced subgraphs, called* expanding cores *or just* cores*, where $t = O((n \log n)/h)$ such that*

- *Each core $K \in \mathcal{F}$ is a φ-expander and $\deg_K(u) \geq \varphi h/3$ for all $u \in V(K)$ where $\varphi = \Omega(1/\log n)$. Moreover, K has diameter $O((\log n)/\varphi)$ and is $(\varphi h/3)$-edge-connected.*
- *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 $\bigcup_{K \in \mathcal{F}} V(K)$.*

Proof: We start with the following two propositions.

Proposition B.2 *Let $G = (V, E)$ be an n-vertex m-edge graph. Then there is a partition $V_1, \ldots, V_k$ of V into disjoint sets, such that $\sum_{i=1}^{k} \delta(V_i) \leq m/2$, and for all $1 \leq i \leq k$, $G[V_i]$ is strong φ-expander w.r.t. G where $\varphi = \Omega(1/\log n)$.*

Proof: The well-known φ-expander decomposition (e.g. Observation 1.1. of [SW19]) says that, given any graph $G = (V, E)$ with m edges (possibly with self-loops and multi-edges) and a parameter φ, there exists a partition $V_1, \ldots, V_k$ of V such that $\sum_{i=1}^{k} \delta_G(V_i) \leq O(\varphi m \log m)$ and $G[V_i]$ is a φ-expander.

Let G' be obtained from G by adding, for each vertex v, $\deg_G(v)$ self-loops at v. We claim that a φ-expander decomposition $V'_1, \ldots, V'_k$ of G' where $\varphi = \Omega(1/\log m)$ is indeed the desired strong expander decomposition for G. This is because, for any set $\emptyset \neq S \subset V_i$, we have $\mathrm{vol}_{G'[V'_i]}(S) \geq \mathrm{vol}_G(S)$ because of the self-loops and $\delta_{G'[V_i]}(S) = \delta_{G[V_i]}(S)$. So we have that $\frac{\delta_{G[V'_i]}(S)}{\min\{\mathrm{vol}_G(S), \mathrm{vol}_G(V'_i \setminus S)\}} \geq \frac{\delta_{G'[V'_i]}(S)}{\min\{\mathrm{vol}_{G'[V'_i]}(S), \mathrm{vol}_{G'[V'_i]}(V_i \setminus S)\}} \geq \varphi$. That is, $G[V'_i]$ is indeed a strong φ-expander with respect to G. Also, for each i, $\delta_G(V'_i) = \delta_{G'}(V'_i)$. So we have $\sum_{i=1}^{k} \delta_G(V'_i) = \sum_{i=1}^{k} \delta_G(V'_i) \leq O(\varphi \cdot (2m) \log(2m)) \leq m/2$ by choosing an appropriate constant in $\varphi = \Omega(1/\log m)$. □

Proposition B.3 *Let H' be an n-vertex graph with minimum degree h'. Then there is a collection $\mathcal{F}'$ of vertex-disjoint induced subgraphs of H' that we call cores, such that:*

- *Each core $K \in \mathcal{F}'$ is a φ-expander and for all $u \in V(K)$, $\deg_K(u) \geq \varphi h'$, where $\varphi = \Omega(1/\log n)$; and*
- $\sum_{K \in \mathcal{F}'} |E(K)| \geq 3|E(H')|/4$.

Proof: We apply Proposition B.2 to graph H' to obtain a partition $(V_1, \ldots, V_k)$ of $V(H')$. We then let $\mathcal{F}$ contain all graphs $H'[V_i]$ with $|V_i| \geq 2$. Notice that, from Proposition B.2, each such graph $H'[V_i]$ is a φ-expander. Moreover, from by Observation 2.1, for all $u \in V_i$, $\deg_{G[V_i]}(u) \geq \varphi h'$ Lastly, observe that $\sum_{K \in \mathcal{F}'} |E(K)| = |E(H')| - (\sum_{i=1}^{k} \delta_{H'}(V_i))/2 \geq 3|E(H')|/4$. □

We are now ready to provide the algorithm for constructing the core decomposition, that will be used in the proof of Theorem B.1.

The algorithm. We start with $\mathcal{F} \leftarrow \emptyset$, $H' \leftarrow H$, and $h' \leftarrow h/3$. Let $A = \textsf{Proc-Degree-Pruning}(H', h')$. We set $H' \leftarrow H'[A]$, so that H' has minimum degree at least h'. Then, we apply Proposition B.3 to H' and obtain the collection $\mathcal{F}'$ of cores. We set $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}'$ and delete all vertices in $\bigcup_{K \in \mathcal{F}'} V(K)$ from H'. Then, we again set $A = \textsf{Proc-Degree-Pruning}(H', h')$ and repeat this process until $H' = \emptyset$. Let $\mathcal{F}$ be the final collection of cores that the algorithm computes. We now prove that it has all required properties.

The first guarantee. Proposition B.3 directly guarantees that each core $K \in \mathcal{F}$ is a φ-expander, and moreover, for all $u \in V(K)$, $\deg_K(u) \geq \varphi h'$. By the standard ball-growing argument, any φ-expander has diameter at most $O(\log(n)/\varphi) = O(\log^2 n)$. Next, to prove that K is $(\varphi h')$-edge connected, it is enough show that, for any vertex set $S \subseteq V(K)$ with $\mathrm{vol}_K(S) \leq \mathrm{vol}(K)/2$, $\delta_K(S) \geq \varphi h'$ holds. Observe that, since K is a φ-expander, $\delta_K(S) \geq \varphi \mathrm{vol}_K(S) \geq \varphi^2 h'|S|$ must hold. At the same time, since the minimum degree in K is at least $\varphi h'$ and K is a simple graph, $\delta_K(S) \geq \varphi h'|S| - \binom{|S|}{2}$ must hold. We now consider two cases. First, if $|S| \geq 1/\varphi$, then $\varphi^2 h'|S| \geq \varphi h'$. Otherwise, it can be verified that $\varphi h'|S| - \binom{|S|}{2} \geq \varphi h'$ for all $1 \leq |S| < 1/\varphi$. In any case, $\delta_K(S) \geq \varphi h'$.

The second guarantee. We denote $U = V(H) \setminus \bigcup_{K \in \mathcal{F}} V(K)$. Note that $v \in U$ only if, for some graph H' that arose over the course of the algorithm, $v \notin A$, where $A = \textsf{Proc-Degree-Pruning}(H', h')$. We say that vertex v was *removed* when procedure Proc-Degree-Pruning was applied to that graph

H'. By orienting edges incident to v towards v whenever v is removed, we can orient all edges of H incident to the vertex set U such that $\text{in-deg}_H(v) \leq h'$ for each $v \in U$. Let $\overrightarrow{H}$ be a directed graph obtained from H by contracting all vertices in $\bigcup_{K \in \mathcal{F}} V(K)$ into a single vertex t, while keeping the orientation of edges incident to U. Observe that $V(\overrightarrow{H}) = U \cup \{t\}$ and $\overrightarrow{H}$ is a DAG with t as a single sink. It is now enough to show that, for every vertex $u \in U$, there are $2h/3$ edge-disjoint directed paths of length $O(\log n)$ in $\overrightarrow{H}$ from u to t.

For any $S \subseteq V(\overrightarrow{H})$, let $\text{in-vol}_{\overrightarrow{H}}(S) = \sum_{u \in S} \text{in-deg}_{\overrightarrow{H}}(u)$, $\text{out-vol}_{\overrightarrow{H}}(S) = \sum_{u \in S} \text{out-deg}_{\overrightarrow{H}}(u)$, and $\text{vol}_{\overrightarrow{H}}(S) = \text{in-vol}_{\overrightarrow{H}}(S) + \text{out-vol}_{\overrightarrow{H}}(S)$. Observe that, for $v \in U$, $\text{out-deg}_{\overrightarrow{H}}(v) \geq 2\text{in-deg}_{\overrightarrow{H}}(v)$ because $\text{in-deg}_{\overrightarrow{H}}(v) \leq h' = h/3$ but $\deg_{\overrightarrow{H}}(v) \geq h$. So, for any $S \subseteq U$, $\text{out-vol}_{\overrightarrow{H}}(S) \geq 2\text{in-vol}_{\overrightarrow{H}}(S)$.

Fix a vertex $u \in U$. Let $B_d = \{v \mid \mathsf{dist}_{\overrightarrow{H}}(u, v) \leq d\}$. Suppose that $B_d \subseteq U$, then we have

$$\begin{aligned}
\text{vol}_{\overrightarrow{H}}(B_{d+1}) &= \text{vol}_{\overrightarrow{H}}(B_d) + \text{vol}_{\overrightarrow{H}}(B_{d+1} \setminus B_d) \\
&\geq \text{vol}_{\overrightarrow{H}}(B_d) + |E_{\overrightarrow{H}}(B_d, B_{d+1} \setminus B_d)| \\
&= \text{vol}_{\overrightarrow{H}}(B_d) + |E_{\overrightarrow{H}}(B_d, B_{d+1})| - |E_{\overrightarrow{H}}(B_d, B_d)| \\
&\geq \text{vol}_{\overrightarrow{H}}(B_d) + \text{out-vol}_{\overrightarrow{H}}(B_d) - \text{in-vol}_{\overrightarrow{H}}(B_d) \\
&\geq \text{vol}_{\overrightarrow{H}}(B_d) + \text{vol}_{\overrightarrow{H}}(B_d)/3 = (4/3)\text{vol}_{\overrightarrow{H}}(B_d)
\end{aligned}$$

where the last inequality is because $\text{out-vol}_{\overrightarrow{H}}(B_d) \geq 2\text{in-vol}_{\overrightarrow{H}}(B_d)$. This proves that $t \in B_{3\log_{4/3} n}$, otherwise $\text{vol}_{\overrightarrow{H}}(B_{3\log_{4/3} n}) \geq (4/3)^{3\log_{4/3} n} \geq n^3$ which is a contradiction. This implies that there is a directed u-t path P of length $O(\log n)$ in $\overrightarrow{H}$, but we want to show that there are many such edge-disjoint paths.

Observe that the argument above only exploits the fact that $\text{out-deg}_{\overrightarrow{H}}(v) \geq 2\text{in-deg}_{\overrightarrow{H}}(v)$ for all $v \in U$. So even if we remove edges of a u-t path P from $\overrightarrow{H}$, this inequality still holds for all $v \in U \setminus \{u\}$. As we can assume that $\text{in-deg}_{\overrightarrow{H}}(u) = 0$ because in-coming edges to u do not play a role for finding u-t paths, we also have $\text{out-deg}_{\overrightarrow{H}}(u) \geq 2\text{in-deg}_{\overrightarrow{H}}(u) = 0$. Therefore, we can repeat the argument $\text{out-deg}_{\overrightarrow{H}}(u) \geq h - h' \geq 2h/3$ times, and obtains $2h/3$ edge-disjoint u-t paths in $\overrightarrow{H}$. So we conclude that, for each vertex $u \in U = V(H) \setminus \bigcup_{K \in \mathcal{F}} V(K)$, there are $2h/3$ edge-disjoint paths of length $O(\log n)$ from u to vertices in $\bigcup_{K \in \mathcal{F}} V(K)$.

□

B.3 Proof of Theorem 3.6: Strong Expander Decomposition

We will use the recent almost-linear time determinstic algorithm for computing a (standard) expander decomposition by Chuzhoy et al. [CGL+19].

Theorem B.4 (Restatement of Corollary 7.7 from [CGL+19]) *There is a deterministic algorithm that, given a graph $G = (V, E)$ with m edges (possibly with self-loops and parallel edges), a parameter $\varphi \in (0, 1)$, and a number $r \geq 1$, computes a partition of V into disjoint subsets $V_1, \ldots, V_k$ such that $\sum_{i=1}^{k} \delta_G(V_i) \leq \varphi m \cdot (\log m)^{O(r^2)}$, and for all $1 \leq i \leq k$, $G[V_i]$ is a φ-expander. The running time of the algorithm is $O(m^{1+O(1/r)+o(1)} \cdot (\log m)^{O(r^2)})$.*

We can now complete the proof of Theorem 3.6 using Theorem B.4. Given an input graph $G = (V, E)$ for Theorem 3.6, we construct a graph G' as follows. We start by setting $G' \leftarrow G$ and, for each vertex $v \in V$, we add $\deg_G(v)$ self-loops to it in G'. We then apply Theorem B.4 to graph G', with parameters φ and $r = \log^{1/4} m$, to obtain a partition of $V(G')$ into disjoint subsets $V_1, \ldots, V_k$ such that $\sum_{i=1}^{k} \delta_{G'}(V_i) \leq (\log |E(G')|)^{O(r^2)} \cdot \varphi \cdot |E(G')|$, and for all $1 \leq i \leq k$, $G'[V_i]$ is a φ-expander.

First, observe that for each i, $\delta_G(V_i) = \delta_{G'}(V_i)$ because G and G' differ only by self-loops. So we have $\sum_{i=1}^{k} \delta_G(V_i) = (\log |E(G')|)^{O(r^2)} \cdot \varphi \cdot |E(G')| \leq \gamma(m)\varphi m$.

Second, observe that, for any set $\emptyset \neq S \subsetneq V_i$, $\mathrm{vol}_{G'[V_i]}(S) \geq \mathrm{vol}_G(S)$ because of the self-loops in G', and $\delta_{G'[V_i]}(S) = \delta_{G[V_i]}(S)$. So we have that $\frac{\delta_{G[V_i]}(S)}{\min\{\mathrm{vol}_G(S), \mathrm{vol}_G(V_i \setminus S)\}} \geq \frac{\delta_{G'[V_i]}(S)}{\min\{\mathrm{vol}_{G'[V_i]}(S), \mathrm{vol}_{G'[V_i]}(V_i \setminus S)\}} \geq \varphi$. That is, for each i, $G[V_i]$ is indeed a strong φ-expander with respect to G. Therefore, we can simply return the partition $\{V_1, \ldots, V_k\}$ as an output for Theorem 3.6. The running time is $O(m^{1+O(1/r)+o(1)} \cdot (\log m)^{O(r^2)}) = \widehat{O}(m)$ by Theorem B.4.

B.4 Proof of Theorem 3.8: Embedding Small Expanders

In this section we prove Theorem 3.8. The proof uses the *cut-matching game*, that was introduced by Khandekar, Rao, and Vazirani [KRV09] as part of their fast randomized algorithm for the Sparsest Cut and Balanced Cut problems. Chuzhoy et al. [CGL+19] provided an efficient deterministic implementation of this game (albeit with weaker parameters), based on a variation of this game due to Khandekar et al. [KKOV07]. We start by describing the variant of the Cut-Matching game that we use, that is based on the results of [CGL+19].

B.4.1 Deterministic Cut-matching Game

The *cut-matching game* is a game that is played between two players, called the *cut player* and the *matching player*. The game starts with a graph W whose vertex set V has cardinality n, and $E(W) = \emptyset$. The game is played in rounds; in each round i, the cut player chooses a partition (A_i, B_i) of V with $|A_i| \leq |B_i|$. The matching player then chooses an arbitrary matchings M_i that matches **every** vertex of A_i to some vertex of B_i. The edges of M_i are then added to W_i, completing the current round. Intuitively, the game terminates once graph W becomes a ψ-expander, for some given parameter ψ. It is convenient to think of the cut player's goal as minimizing the number of rounds, and of the matching player's goal as making the number of rounds as large as possible. We prove the following theorem, that easily follows from [CGL+19].

Theorem B.5 (Deterministic Algorithm for Cut Player) *There is a deterministic algorithm, that, for every round $i \geq 1$, given the graph W that serves as input to the ith round of the cut-matching game, produces, in time $O(n\gamma(n))$, a partition (A_i, B_i) of V with $|A_i| \leq |B_i|$, such that, no matter how the matching player plays, after $R = O(\log n)$ rounds, the resulting graph W is a $1/\gamma(n)$-expander.*

Proof: For the sake of the proof, it is more convenient to work with the notion of *sparsity* instead of *conductance*.

Definition. (Sparsity) *The sparsity $\Psi(G)$ of a graph $G = (V, E)$ is the minimum, over all vertex sets $S \subseteq V$ with $1 \leq |S| \leq |V \setminus S|$, of $\delta(S)/|S|$.*

From the definition, it is immediate to see that, if a graph G has maximum degree d, then $\Phi(G) \leq \Psi(G) \leq d \cdot \Phi(G)$. In particular, if $\Psi(G) \geq \varphi'$ for any parameter φ', then G is a (φ'/d)-expander. Clearly, for any subgraph $H \subseteq G$ with $V(H) = V(G)$, $\Psi(H) \leq \Psi(G)$ must hold. We need the following observation:

Observation B.6 (Observation 2.3 of [CGL+19]) *Let $G = (V, E)$ be an n-vertex graph with $\Psi(G) \geq \psi$, and let G' be another graph that is obtained from G by adding to it a new set V' of*

at most n vertices, and a matching M connecting every vertex of V' to a distinct vertex of G. Then $\Psi(G') = \Omega(\psi)$.

In order to implement the algorithm of the cut player, we will employ the following algorithm by [CGL+19]:

Theorem B.7 (Theorem 1.6 of [CGL+19]) *There is a deterministic algorithm, called* CutOrCertity*, that, given an n-vertex graph $G = (V, E)$ with maximum vertex degree $O(\log n)$, and a parameter $r \geq 1$, returns one of the following:*

- *either a cut (A, B) in G with $n/4 \leq |A| \leq |B|$ and $|E_G(A, B)| \leq n/100$; or*
- *a subset $S \subseteq V$ of at least $n/2$ vertices, such that $\Psi(G[S]) \geq 1/\log^{O(r)} n$.*

The running time of the algorithm is $O(n^{1+O(1/r)} \cdot (\log n)^{O(r^2)})$.

The following lemma (first proved by [KKOV07]) shows that it is impossible for the cut player to return a balanced sparse cut for more than $O(\log n)$ iterations of the cut-matching game.

Lemma B.8 (Restatement of Theorem 2.5 of [CGL+19]) *There is a constant c, for which the following holds. Consider the cut-matching game where in each iteration $1 \leq i \leq c \log n$, we use Algorithm* CutOrCertity *in order to implement the cut player; specifically, if the algorithm returns a partition (A_i, B_i) of V with $|E_W(A_i, B_i)| \leq n/100$ and $n/4 \leq |A_i| \leq |B_i|$, then we let (A'_i, B'_i) be any partition of V with $A'_i \subseteq A_i$ and $|A'_i| = |B'_i|$, and we use the partition (A'_i, B'_i) as the response of the cut player in round i. Otherwise, we terminate the cut-matching game. Then no matter how the matching player plays in each iteration, the game will be terminated before reaching iteration $\lfloor c \log n \rfloor$.*

We are now ready to complete the proof of Theorem B.5. In each iteration $1 \leq i \leq O(\log n)$ of the cut-matching game, we apply algorithm CutOrCertity to graph W_{i-1} (where W_0 is the initial graph with $E(W) = \emptyset$), with the parameter $r = O(\log^{1/4} n)$. Since the edge set $E(W_{i-1})$ is the union of $i - 1$ matchings, graph W_{i-1} it has maximum degree at most $i - 1$. We will ensure that the number of rounds is bounded by $O(\log n)$, so graph W_{i-1} is a valid input for CutOrCertity.

If CutOrCertity returns a cut (A_i, B_i) with $E_{W_{i-1}}(A_i, B_i) \leq n/100$ and $n/4 \leq |A_i| \leq |B_i|$, then we output an arbitrary partition (A'_i, B'_i) of V with $A'_i \subseteq A_i$ and $|A'_i| = |B'_i|$. By Lemma B.8, this can happen for at most $O(\log n)$ iterations, regardless of the responses of the matching player. Otherwise, if CutOrCertity returns a subset $S \subseteq V$ of at least $n/2$ vertices, with $\Psi(W_{i-1}[S]) \geq 1/\log^{O(r)} n$, we output the partition (A_i, B_i), where $A_i = V \setminus S$ and $B_i = T$. Let M_i be the matching returned by the matching player, that matches every vertex of $V \setminus S$ to a distinct vertex of S. We are then guaranteed that the graph $W_i = W_{i-1} \cup M_i$ is a φ'-expander, where $\varphi' = 1/\log^{O(r)} n = 1/(\log n)^{O(\log^{1/4} n)} \geq \gamma(n)$ (recall that $\gamma(n) = \exp(\log^{3/4} n)$; we have also used the fact that the maximum vertex degree in W_i is at most $O(\log n)$). We conclude that W_i is a $(1/\gamma(n))$-expander.

The running time of the algorithm is dominated by Algorithm CutOrCertity, whose running time is $O(n^{1+O(1/r)} \cdot (\log n)^{O(r^2)}) = O(n\gamma(n))$. This completes the proof of Theorem B.5. □

In order to complete the proof of Theorem 3.8, we next provide an efficient deterministic algorithm for the cut player. The idea (that is quite standard) is that, in addition to producing the required matching M_i in each round of the game, the cut player will also embed the edges of M_i into the graph G, where the embedding paths have a relatively short length and cause a relatively small congestion.

B.4.2 Implementing the Matching Player

There are well-known efficient algorithms, that, given a φ-expander $G = (V, E)$, and any two vertex subsets $A, B \subseteq V$, compute a large collection of paths between vertices of A and vertices of B, that cause congestion $\tilde{O}(1/\operatorname{poly}(\varphi))$, such that every path has length $\tilde{O}(1/\operatorname{poly}(\varphi))$. We will use such an algorithm in order to implement the matching player. The algorithm is summarized in the following theorem, that uses the approach of [CK19, CGL+19]. We include the proof for completeness.

Theorem B.9 *There is a deterministic algorithm, that we call* TerminalMatching(G, A, B, φ), *that receives as input a parameter $\varphi \in (0, 1)$, a φ-expander $G = (V, E)$ with m edges, and two subsets $A, B \subseteq V$ of vertices of G called terminals, with $|A| \leq |B|$. The algorithm returns a matching M between vertices of A and vertices of B of cardinality $|A|$, and a set $\mathcal{P}$ of paths of length at most $O(\log n/\varphi)$ each, embedding the matching M into G with edge-congestion $O(\log^2 n/\varphi^2)$. The running time of the algorithm is $\tilde{O}(m/\varphi^3)$.*

The remainder of this subsection is dedicated to proving Theorem B.9. The proof is almost identical to that in [CGL+19]. The key subroutine that is used in the proof is the following lemma.

Lemma B.10 *There is a deterministic algorithm, that, given an m-edge graph $G = (V, E)$, two disjoint subsets A', B' of its vertices with $|A'| \leq |B'|$, and an integer $\ell \geq 32 \log m$, computes one of the following:*

- *either a matching M' between vertices of A' and vertices of B' with $|M'| \geq |A'| \cdot \frac{8 \log m}{\ell^2}$, together with a collection $\mathcal{P}'$ paths of length at most ℓ each that embed M' into G with edge-congestion 1; or*
- *a cut (X, Y) in G with $\Phi_G(X, Y) \leq 24 \log m/\ell$.*

The running time of the algorithm is $\tilde{O}(\ell|E(G)|)$.

Proof: We can assume w.l.o.g. that the graph G is connected, as otherwise we can compute a cut (X, Y) with conductance zero in time $O(m)$. Next, we create an auxiliary graph G_{st} as follows. We start with graph G, and then add a source vertex s that connects with an edge to every vertex of A', and a sink vertex t, that connects with an edge to every vertex of B'. We then initialize an ES-Tree data structure on graph $G_{s,t}$, with source vertex s, and distance threshold $\ell + 2$. We denote by $\mathcal{T}$ the shortest-path tree rooted at s that the data structure maintains. We also initialize $\mathcal{P}' = \emptyset$ and $M' = \emptyset$.

The algorithm performs iterations, as long as $\mathsf{dist}_{G_{st}}(s, t) \leq \ell + 2$ and $|\mathcal{P}'| < \frac{8|A'| \log m}{\ell^2}$ hold.

In order to execute an iteration, let P_{st} be the shortest s-t path in $\mathcal{T}$. Observe that path $P' = P_{st} \setminus \{s, t\}$ is a simple path in graph G, of length at most ℓ, connecting some vertex $a' \in A'$ to some vertex $b' \in B'$. We delete the edges of P_{st} from G_{st} and update the ES-Tree data structure accordingly. Also, we add the path $P_{st} \setminus \{s, t\}$ to set $\mathcal{P}'$ and set $A' \leftarrow A' \setminus \{a'\}$ and $B' \leftarrow B' \setminus \{b'\}$. Lastly, we add (a', b') to M, and continue to the next iteration.

Notice that the total running time of the algorithm so far is $O(m\ell)$, by the guarantees of the ES-Tree data structure.

We now consider two cases. First, if, when the above algorithm terminates, $|\mathcal{P}'| = \frac{8|A'| \log m}{\ell^2}$ holds, then we return the matching M' and the paths set $\mathcal{P}'$. Clearly, $|M'| \geq |A'| \cdot \frac{8 \log m}{\ell^2}$ holds, the paths in $\mathcal{P}'$ are edge-disjoint (so they cause edge-congestion 1), and the length of every path is at most ℓ.

Therefore, we assume from now on that, when the above algorithm terminates, $|\mathcal{P}'| < \frac{8|A'|\log m}{\ell^2}$ holds, and so, from the algorithm description, $\mathsf{dist}_{G_{st}}(s,t) > \ell+2$. In particular, $\mathsf{dist}_{G_{st}}(A',B') > \ell$ now holds, where $\mathsf{dist}_{G_{st}}(A',B') = \min_{a'\in A', b'\in B'} \mathsf{dist}_{G_{st}}(a',b')$. We use the following standard claim:

Claim B.11 *There is a deterministic algorithm, that, given an m-edge graph H with two sets S,T of its vertices, such that $\mathsf{dist}_H(S,T) > \ell$ for some parameter ℓ, computes a vertex set Z with $\Phi_H(Z) < \frac{8\log m}{\ell}$, and $\mathrm{vol}_H(Z) \leq \mathrm{vol}(H)/2$, such that either $S \subseteq Z$ or $T \subseteq Z$ hold. The running time of the algorithm is $O(m)$.*

Proof: For any vertex set $X \subseteq V(H)$ and a parameter d, let $\mathrm{Ball}_H(X,d) = \{u \mid \mathsf{dist}_H(X,u) \leq d\}$. Note that we are guaranteed that $\mathrm{Ball}_H(S,\ell/3) \cap \mathrm{Ball}_H(T,\ell/3) = \emptyset$. Therefore, we can assume w.l.o.g. that $\mathrm{vol}(\mathrm{Ball}_H(S,\ell/3)) \leq \mathrm{vol}(H)/2$. We claim that there must be an index $0 \leq i \leq \ell/3$ such that $\delta(\mathrm{Ball}_H(S,i)) \leq \frac{8\log m}{\ell} \cdot \mathrm{vol}(\mathrm{Ball}_H(S,i))$. Indeed, assume otherwise. Then $\mathrm{vol}(\mathrm{Ball}_H(S,\ell/3)) \geq (1+\frac{8\log m}{\ell})^{\ell/3} > \mathrm{vol}(H)/2$ which is a contradiction. We can compute the index i and the vertex set $Z = \mathrm{Ball}_H(S,i)$ by performing breadth-first search from vertices of S and vertices of T, in time $O(m)$. It is now easy to verify that $\Phi_H(Z) < \frac{8\log m}{\ell}$, $\mathrm{vol}_H(Z) \leq \mathrm{vol}(H)/2$, and either $S \subseteq Z$ or $T \subseteq Z$ hold. □

We are now ready to complete the proof of Lemma B.10. Let $H = G_{st} \setminus \{s,t\}$. We invoke Claim B.11 on graph H, with the sets A' and B' of vertices, and obtain a cut Z. We claim that $\Phi_G(Z) < \frac{24\log m}{\ell}$. Indeed, let E_1 denote the set of all edges lying on the paths in $\mathcal{P}'$, and let E_2 denote the set of all edges in $\delta_G(Z) \setminus E_1$. From the guarantees of Lemma B.10, $|E_2| \leq \frac{8\log m}{\ell} \cdot \mathrm{vol}_H(Z) \leq \frac{8\log m}{\ell} \cdot \mathrm{vol}_G(Z)$. Let k denote the cardinality of the set A' at the beginning of the algorithm. Since $|\mathcal{P}'| < \frac{8k\log m}{\ell^2} < \frac{k}{2}$ (as we have assumed that $\ell \geq 32\log m$), we get that $|A'| \geq k/2$, and in particular, $\mathrm{vol}_G(Z) \geq |A'| \geq k/2$.

In order to complete the proof that $\Phi_G(Z) < \frac{24\log m}{\ell}$, it is now enough to show that $|E_1| \leq \frac{16\log m}{\ell} \cdot \mathrm{vol}_G(Z)$. Indeed, recall that the length of every path in $\mathcal{P}'$ is bounded by ℓ, and $|\mathcal{P}'| < \frac{8k\log m}{\ell^2}$. Therefore:

$$|E_1| \leq \ell \cdot |\mathcal{P}'| < \frac{8k\log m}{\ell}.$$

Since we have established that $\mathrm{vol}_G(Z) \geq k/2$, we get that $|E_1| < \frac{16\log m}{\ell}\mathrm{vol}_G(Z)$. We conclude that $E_G(Z) \leq \frac{24\log m}{\ell}\mathrm{vol}_G(Z)$, as required.

As the total running time of the algorithm is bounded by $O(m\ell)$, this concludes the proof of Lemma B.10. □

We obtain the following corollary of Lemma B.10.

Corollary B.12 *There is a deterministic algorithm, that, given an m-edge graph $G=(V,E)$, two disjoint subsets A,B of its vertices with $|A| \leq |B|$, and a parameter $\ell \geq 32\log m$, computes one of the following:*

- *either a matching M between vertices of A and vertices of B with $|M| = |A|$, and a collection $\mathcal{P}$ of paths of length at most ℓ each, that embeds M into G with congestion at most ℓ^2; or*
- *a cut (X,Y) in G where $\Phi_G(X,Y) \leq 24\log m/\ell$.*

The running time of the algorithm is $\tilde{O}(\ell^3 m)$.

Proof: We start with $M = \emptyset$ and $\mathcal{P} = \emptyset$, and then iterate. In every iteration, we let $A' \subseteq A$ and $B' \subseteq B$ be the subsets of vertices that do not participate in the matching M; since $|A| \leq |B|$, we

are guaranteed that $|A'| \leq |B'|$. If $A' = \emptyset$, then we terminate the algorithm, and return the current matchings M and its embedding $\mathcal{P}$ that we have computed. Otherwise, we apply the algorithm from Lemma B.10 to graph G and vertex sets A', B'. If the outcome is a cut (X, Y) in G with $\Phi_G(X, Y) \leq 24 \log m/\ell$, then we terminate the algorithm, and return the cut (X, Y). Therefore, we assume from now on that, whenever Lemma B.10 is called, it returns a matching M' between A' and B' with $|M'| \geq \frac{8|A'| \log m}{\ell^2}$, and its corresponding embedding $\mathcal{P}'$ with congestion 1 and length ℓ. We then add the paths in $\mathcal{P}'$ to $\mathcal{P}$, and we add the matching M' to M, and continue to the next iteration.

As $|M'| \geq \frac{8|A'| \log m}{\ell^2}$ in every iteration, after ℓ^2 iterations, we must have $A' = \emptyset$, and the algorithm will terminate. Notice that the congestion of the final path set $\mathcal{P}$ is bounded by the number of iterations, ℓ^2. Moreover, since the running time of each iteration is $\tilde{O}(\ell m)$, the total running time of the algorithm is $\tilde{O}(\ell^3 m)$. □

We are now ready to complete the proof of Theorem B.9. We set $\ell = 32 \log m/\varphi$, and then run the algorithm from Corollary B.12 on graph G, with vertex sets A and B. As graph G is a φ-expander, the algorithm for Theorem B.9 may never return a cut (X, Y) with $\Phi_G(X, Y) \leq 24 \log m/\ell < \varphi$. Therefore, the algorithm must return a matching M between the vertices of A and the vertices of B of cardinality $|A|$, together with its embedding $\mathcal{P}$, whose congestion is $\ell^2 = O(\log^2 m/\varphi^2)$, such that the length of every path in $\mathcal{P}$ is bounded by $\ell = O(\log m/\varphi)$. The total running time of the algorithm is $\tilde{O}(\ell^3 m) = \tilde{O}(m/\varphi^3)$.

B.4.3 Completing the Proof of Theorem 3.8.

We are now ready to complete the proof of Theorem 3.8. We run the cut-matching game on a graph W whose vertex set is the set T of terminals, and the edge set is initially empty. In every round i of the game, we use the algorithm from Theorem B.5 to compute a partition (A_i, B_i) of T with $|A_i| \leq |B_i|$, that we treat as the move of the cut player. Then we apply Algorithm TerminalMatching from Theorem B.9 to graph G, and the sets A_i and B_i of vertices. We denote by M_i the matching returned by the algorithm, and by $\mathcal{P}_i$ its embedding. We then add the edges of M_i to graph W and continue to the next iteration. From Theorem B.5, after $O(\log |T|)$ iterations, graph W is guaranteed to be a $(1/\gamma(|T|))$-expander. Since the edge set $E(W)$ is a union of $O(\log |T|)$ matchings, every vertex of W has degree at most $O(\log |T|)$. We also compute an embedding $\mathcal{P} = \bigcup_i \mathcal{P}_i$ of W into G, where every path in $\mathcal{P}$ has length $O(\log(n)/\varphi)$. Moreover, since each path set $\mathcal{P}_i$ causes edge congestion at most $O(\log^2(n)/\varphi^2)$, the congestion of the embedding $\mathcal{P}$ is at most $O(\log^3(n)/\varphi^2)$. Lastly, it remains to bound the running time of the algorithm. Recall that the algorithm consists of $O(\log n)$ rounds. In every round we apply the algorithm from Theorem B.5, whose running time is $O(n\gamma(n))$, and Algorithm TerminalMatchingfrom Theorem B.9, whose running time is $\tilde{O}(m/\varphi^3)$. Therefore, the total running time of the algorithm is bounded by $\tilde{O}\left(m/\varphi^3 + |T|\gamma(|T|)\right) = \tilde{O}\left(m\gamma(|T|)/\varphi^3\right)$. This concludes the proof of Theorem 3.8.

C Application: Maximum Bounded-Cost Flow

In this section, we provide an algorithm for the Maximum Bounded-Cost Flow problem, as the main application of our algorithm for decremental SSSP from Theorem 1.1. The technique is a standard application of the multiplicative weight update framework [GK98, Fle00, AHK12]. We provide the proofs for completeness. In [CK19], the same technique was used to provide algorithms for Maximum s-t Flow in vertex-capacitated graphs. We note that the Maximum Bounded-Cost Flow problem is somewhat more general, and it is a useful subroutine for a large number of applications; we discuss some of these applications in Appendix D. We start with some basic definitions.

Definitions. Given a directed graph $G = (V, E)$ and a pair $s, t \in V$ of its vertices, an s-t flow is a function $f \in \mathbb{R}^E_{\geq 0}$, such that, for every vertex $v \in V - \{s, t\}$, the amount of flow entering v equals the amount of flow leaving v, that is, $\sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)\in E} f(v, u)$. Let $f(v) = \sum_{(u,v)\in E} f(u, v)$ be the *amount of flow at* v. The *value* of the flow f is $\sum_{(s,v)\in E} f(s, v) - \sum_{(v,s)\in E} f(v, s)$. Assume further that we are given capacities $c \in (\mathbb{R}_{>0} \cup \{\infty\})^E$ and costs $b \in \mathbb{R}^E_{\geq 0}$ on edges. A flow f is *edge-capacity-feasible* if $f(e) \leq c(e)$ for all $e \in E$. The *cost* of the flow f is $\sum_{e\in E} b(e)f(e)$. If we are given a cost bound $\bar{b} \geq 0$, then we say that f is *edge-cost-feasible* iff $\sum_{e\in E} b(e)f(e) \leq \bar{b}$. We can define capacities and costs on the vertices of G similarly. Let $c \in (\mathbb{R}_{>0} \cup \{\infty\})^V$ be vertex capacities and let $b \in \mathbb{R}^V_{\geq 0}$ be vertex costs. As before, we say that f is *vertex-capacity-feasible* if $f(v) \leq c(v)$ for all $v \in V - \{s, t\}$, and it is *vertex-cost-feasible* if $\sum_{v\in V-\{s,t\}} b(v)f(v) \leq \bar{b}$. We may just write *capacity-feasible* and *cost-feasible* when clear from context. If G is undirected, one way to define an s-t flow is by treating G as a directed graph, where we replace each undirected edge $\{u, v\}$ with a pair (u, v) and (v, u) of bi-directed edges. We will assume w.l.o.g. that the flow only traverses the edges in one direction, that is, for each edge $\{u, v\} \in E$, either $f(u, v) = 0$ or $f(v, u) = 0$.

Next, we define the *Maximum Bounded-Cost Flow problem* (MBCF). In the edge-capacitated version, we are given a graph $G = (V, E)$ with edge capacities $c \in (\mathbb{R}_{>0} \cup \{\infty\})^E$ and edge costs $b \in \mathbb{R}^E_{\geq 0}$, together with two special vertices s and t, and a cost bound $\bar{b}$. The goal is to compute an s-t flow f of maximum value, such that f is both capacity-feasible and cost-feasible. The vertex-capacitated version is defined similarly except that we are given vertex capacities $c \in (\mathbb{R}_{>0} \cup \{\infty\})^V$ and vertex cost $b \in \mathbb{R}^V_{\geq 0}$ instead. A $(1 + \epsilon)$-approximate solution for this problem is a flow f which is both capacity-feasible and cost-feasible, such that the value of the flow is at least $\mathsf{OPT}(\bar{b})/(1 + \epsilon)$ where $\mathsf{OPT}(\bar{b})$ is the maximum value of a capacity-feasible flow of cost at most $\bar{b}$.

Connection to the Min-Cost Flow Problem. The classical Min-Cost Flow problem is defined exactly like MBCF, except that, instead of the cost bound, we are given a target flow value τ. The goal is to either (i) compute an s-t flow f of value at least τ, such that f is capacity-feasible and has the smallest cost among all flows satisfying these requirements, or (ii) to certify that there is no capacity-feasible flow of value at least τ. Let $\mathsf{OPT}_{\mathsf{cost}}(\tau)$ be the minimum cost of any capacity-feasible flow of value at least τ. Observe that an exact algorithm for MBCF implies an exact algorithm for the min-cost flow problem and vice versa, via binary search. Moreover, a $(1+\epsilon)$-approximation algorithm for MBCF gives a $(1 + \epsilon)$-factor pseudo-approximation for the Min-Cost Flow problem, that is: we either find a flow of cost at most $\mathsf{OPT}_{\mathsf{cost}}(\tau)$ and value at least $\tau/(1 + \epsilon)$, or certify that there is no capacity-feasible flow of value at least τ. Note that, if we insist that the value of the flow that we obtain in the min-cost flow is at least τ, then the problem is at least as difficult as the **exact** maximum s-t flow. From now on we focus on the MBCF problem.

Our results. We show approximation algorithms for the MBCF problem in undirected graphs for both edge-capacitated and vertex-capacitated settings, though in the former scenario we only consider unit edge-capacities.

Theorem C.1 (Unit-edge capacities) *There is a deterministic algorithm that, given a simple undirected n-vertex m-edge graph $G = (V, E)$ with unit edge capacities $c(e) = 1$ and edge costs $b(e) > 0$ for $e \in E$, together with a source s, a sink t, a cost bound $\bar{b}$, and an accuracy parameter $0 < \epsilon < 0.1$, computes a $(1 + \epsilon)$-approximate solution for MBCF in time $\widehat{O}\left(n^2 \cdot \frac{\log B}{\epsilon^{O(1)}}\right)$, where B is the ratio of largest to smallest edge cost.*

Previously, Cohen et al. [CMSV17] gave an exact algorithm for MBCF with running time $\tilde{O}(m^{10/7} \cdot \log B)$ when the input graph G has unit edge-capacities as well, but G can be directed and is not neces-

sarily simple. Lee and Sidford [LS14] showed an exact algorithm with running time $\tilde{O}(m\sqrt{n}\cdot\log^{O(1)} B)$ on directed graphs with general edge-capacities. To the best of our knowledge, no faster algorithms for MBCF are currently known, even when approximation is allowed. Our algorithm provides a $(1+\epsilon)$-approximate solution, and only works in simple, undirected graphs with unit edge-capacities. It is faster than these previously known algorithms when $m = \omega(n^{1.5+o(1)})$, and it implies a number of applications, as shown in Appendix D.

We also show a deterministic algorithm for graphs with (arbitrary) vertex capacities and costs.

Theorem C.2 (Vertex capacities) *There is a deterministic algorithm that, given an undirected n-vertex graph $G = (V, E)$ with vertex capacities $c(v) > 0$ and vertex costs $b(v) > 0$ for all $v \in V$, a source s, a sink t, a cost bound $\bar{b}$, and an accuracy parameter $0 < \epsilon < 0.1$, computes a $(1+\epsilon)$-approximate* MBCF *in time $\widehat{O}\left(n^2 \cdot \frac{\log(BC)}{\epsilon^{O(1)}}\right)$, where B is the ratio of largest to smallest vertex cost, and C is the ratio largest to smallest vertex capacity.*

We note that a randomized algorithm with similar guarantees can be obtained from the algorithm of Chuzhoy and Khanna [CK19] for SSSP, though this was not explicitly noted in their paper (they only explicitly provide an algorithm for approximate max flow). We obtain a deterministic algorithm by using our deterministic algorithm for SSSP instead of the randomized algorithm of [CK19]. The best previous algorithm for vertex-capacitated MBCF, with running time $\tilde{O}(m\sqrt{n}\log^{O(1)}(BC))$, follows from the work of Lee and Sidford [LS14]; the algorithm solves the problem exactly. Our algorithm has a faster running time when $m = \omega(n^{1.5+o(1)})$.

The remainder of this section is dedicated to proving Theorem C.1 and Theorem C.2. We start by describing, in Appendix C.1, an algorithm for MBCF in general edge-capacitated graphs, based on the multiplicative weight update (MWU) framework, and we bound the number of "augmentations" in the algorithm. Then, in Appendix C.2, we show how to perform the "augmentations" efficiently when the input graph is as in Theorem C.1 and Theorem C.2, using our algorithm for decremental SSSP. We will use the following observation:

Remark C.3 *It is easy to extend Theorem 1.1 so that the algorithm can handle edges of length 0, if we have a promise that in every query* dist-query(s, v) *or* path-query(s, v)*, the distance from the source s to the query vertex v is non-zero. To do this, let $\ell_{\min}$ and $\ell_{\max}$ be the minimum and the maximum* non-zero *edge lengths in the graph respectively. For each edge of length 0, we set the length to be $\epsilon\ell_{\min}/n$ instead. This will not increase the length of any non-zero length path by more than a factor $(1+\epsilon)$. Let $L' = \frac{\ell_{\max}}{\ell_{\min}}$ be the ratio between the original largest to smallest* non-zero *length. We can now use Theorem 1.1 with the new bound $L = L'n/\epsilon$.*

C.1 A Multiplicative Weight Update-Based Flow Algorithm

We describe an algorithm for computing approximate MBCF in edge-capacitated graphs in Algorithm 4. The algorithm is based on the multiplicative weight update (MWU) framework, and it is a straightforward adaptation of the algorithms of Garg and Könemann [GK98], Fleischer [Fle00], and Madry [Mad10].

Algorithm 4 is stated for both undirected and directed graphs. Let $G = (V, E)$ be the input graph and let $\mathcal{P}_{s,t}$ be the set of all s-t paths. If G is (un)directed, then $\mathcal{P}_{s,t}$ contains all (un)directed s-t paths. We always augment a flow along some s-t path $P \in \mathcal{P}_{s,t}$. Let $f(P)$ denote the amount of flow through path P. We use the shorthand $f(P) \leftarrow f(P) + c$ to indicate that we increase the flow value $f(e)$ for all $e \in E(P)$ by c, that is, $f(e) \leftarrow f(e) + c$. The algorithm maintains lengths $\ell \in \mathbb{R}^E_{\geq 0}$ on edges and a parameter $\varphi \geq 0$. For any path P, let $\ell(P) = \sum_{e\in P} \ell(e)$ be the length of P and similarly let

Algorithm 4 An approximate algorithm for max bounded-cost s-t flow in edge-capacitated graphs

Input: An undirected or a directed graph $G = (V, E)$ with edge capacities $c \in (\mathbb{R}_{>0} \cup \{\infty\})^E$ and edge costs $b \in \mathbb{R}^E_{\geq 0}$, a source s, a sink t, a cost bound $\bar{b}$, and an accuracy parameter $0 < \epsilon < 1$.
Output: An s-t flow which is capacity-feasible and cost-feasible.

1. Set $\delta = (2m)^{-1/\epsilon}$
2. Set $\ell(e) = \delta/c(e)$ if $c(e)$ is finite; otherwise $\ell(e) = 0$. Set $\varphi = \delta/\bar{b}$. Set $f \equiv 0$.
3. **while** $\sum_{e \in E} c(e)\ell(e) + \bar{b}\varphi < 1$ **do**
   (a) $P \leftarrow$ a $(1+\epsilon)$-approximate $(\ell + \varphi b)$-shortest s-t path
   (b) $c \leftarrow \min\{\min_{e \in P} c(e), \bar{b}/b(P)\}$
   (c) $f(P) \leftarrow f(P) + c$
   (d) for every edge $e \in E(P)$, set $\ell(e) \leftarrow \ell(e)(1 + \frac{\epsilon c}{c(e)})$
   (e) $\varphi \leftarrow \varphi(1 + \frac{\epsilon c \cdot b(P)}{\bar{b}})$
4. **return** the scaled down flow $f / \log_{1+\epsilon}(\frac{1+\epsilon}{\delta})$

$b(P) = \sum_{e \in P} b(e)$ be the cost of P. In general, for any function $d \in \mathbb{R}^E_{\geq 0}$, we let $d(P) = \sum_{e \in P} d(e)$. We use the shorthand $d = \ell + \varphi b$ to indicate a new edge-length function $d(e) = \ell(e) + \varphi b(e)$ for all $e \in E$. A d-shortest s-t path is an s-t path P^* that minimizes $d(P^*)$ among all paths in $\mathcal{P}_{s,t}$. An α-approximate d-shortest path $\tilde{P}$ is a path $P \in \mathcal{P}_{s,t}$ with $d(\tilde{P}) \leq \alpha \cdot d(P^*)$.

Lemma C.4 *The flow $f / \log_{1+\epsilon}(\frac{1+\epsilon}{\delta})$ computed by Algorithm 4 is capacity-feasible and cost-feasible.*

Proof: When the flow on an edge e is increased by an additive amount of $a \cdot c(e)$, where $0 \leq a \leq 1$, then $\ell(e)$ is multiplicatively increased by factor $(1 + a\epsilon) \geq (1+\epsilon)^a$. As $\ell(e) = \delta/c(e)$ initially and $\ell(e) < (1+\epsilon)/c(e)$ at the end, it grows by the multiplicative factor at most $(1+\epsilon)/\delta = (1+\epsilon)^{\log_{1+\epsilon}((1+\epsilon)/\delta)}$ over the course of the algorithm. Therefore, the flow on e is at most $c(e) \cdot \log_{1+\epsilon}(\frac{1+\epsilon}{\delta})$ before scaling down[9], and so $f / \log_{1+\epsilon}(\frac{1+\epsilon}{\delta})$ is capacity-feasible. Similarly, every time the cost of the flow increases by additive amount $a\bar{b}$, where $0 \leq a \leq 1$, the value of φ is multiplicatively increased by factor $(1 + a\epsilon) \geq (1+\epsilon)^a$. As $\varphi = \delta/\bar{b}$ initially and $\varphi < (1+\epsilon)/\bar{b}$ at the end, the value of φ grows by at most factor $(1+\epsilon)/\delta$ over the course of the algorithm. Therefore, the cost of the final flow f before the scaling down is at most $\bar{b} \cdot \log_{1+\epsilon}(\frac{1+\epsilon}{\delta})$, and so flow $f / \log_{1+\epsilon}(\frac{1+\epsilon}{\delta})$ is cost-feasible. □

Next, we bound the number of augmentation in Algorithm 4, that is, the number of times that the "while" loop is executed.

Lemma C.5

1. *If graph G has unit edge capacities, and there is an s-t cut of capacity k, then there are at most $\tilde{O}(k/\epsilon^2)$ augmentations. In particular, if G is a simple graph with unit edge-capacities, then there are at most $\tilde{O}(n/\epsilon^2)$ augmentations.*
2. *If graph G is has at most k edges with finite capacity, then are at most $\tilde{O}(k/\epsilon^2)$ augmentations.*

[9] If G is undirected, it can be the case that $f(e)$ is even decreased while $\ell(e)$ is increased. This gives even more slack to the analysis.

Proof: (1) By assumption, there is an s-t cut $(S, \overline{S})$ with $|E(S, \overline{S})| = k$. In each augmentation, either φ is increased by factor $(1+\epsilon)$ or there is some edge $e \in E(S, \overline{S})$, for which $\ell(e)$ is increased by factor $(1+\epsilon)$. Again, we have $\ell(e) = \delta/c(e)$ initially and $\ell(e) < (1+\epsilon)/c(e)$ at the end. Also, $\varphi = \delta/\overline{b}$ initially and $\varphi < (1+\epsilon)/\overline{b}$ at the end. So there can be at most $(k+1)\log_{1+\epsilon}(\frac{1+\epsilon}{\delta}) = \tilde{O}(k/\epsilon^2)$ augmentations.

(2) Let E' be the set of edges with finite capacity. In each augmentation, either φ is increased by factor $(1+\epsilon)$ or there is some edge $e \in E'$, for which $\ell(e)$ is increased by factor $(1+\epsilon)$. As before, we start with $\ell(e) = \delta/c(e)$, and at the end, $\ell(e) < (1+\epsilon)/c(e)$ holds. Similarly, $\varphi = \delta/\overline{b}$ initially and $\varphi < (1+\epsilon)/\overline{b}$ holds at the end. Since the total number of edges with finite capacity is at most k, the total number of augmentations is bounded by $(k+1)\log_{1+\epsilon}(\frac{1+\epsilon}{\delta}) = \tilde{O}(k/\epsilon^2)$. □

Lemma C.6 *Flow $f/\log_{1+\epsilon}(\frac{1+\epsilon}{\delta})$ is a $(1+O(\epsilon))$-approximate solution to the* MBCF *problem.*

Proof: For each edge e, let $\mathcal{P}_e \subseteq \mathcal{P}_{s,t}$ be the set of all paths containing e. We first write the standard LP relaxation for MBCF and its dual LP (we can use the same relaxation for both undirected and directed graphs, except that the set $\mathcal{P}_{s,t}$ of paths is defined differently)

$(\boldsymbol{P}_{LP})$					$(\boldsymbol{D}_{LP})$				
max	$\sum_{P\in\mathcal{P}_{st}} f(P)$				min	$\sum_{e\in E} c(e)\ell(e) + \overline{b}\varphi$			
s.t.	$\sum_{P\in\mathcal{P}_e} f(P)$	$\leq$	$c(e)$	$\forall e \in E$	s.t.	$\ell(P) + \varphi b(P)$	$\geq$	1	$\forall P \in \mathcal{P}_{s,t}$
	$\sum_{P\in\mathcal{P}_{s,t}} b(P)\cdot f(P)$	$\leq$	$\overline{b}$			ℓ, φ	$\geq$	0	
	x	$\geq$	0						

Denote $D(\ell, \varphi) = \sum_{e\in E} c(e)\ell(e) + \overline{b}\varphi$, and let $\alpha(\ell, \varphi)$ be the length of the $(\ell + \varphi b)$-shortest s-t path. Let ℓ_i be the edge-length function ℓ after the i-th execution of the while loop, and let φ_i be defined similarly for φ. We denote $D(i) = D(\ell_i, \varphi_i)$ and $\alpha(i) = \alpha(\ell_i, \varphi_i)$ for convenience. We also denote by P_i the path found in the i-th iteration and by c_i the amount by which the flow on P_i is augmented. Observe that:

$$\begin{aligned} D(i) &= \sum_{e\in E} c(e)\ell_{i-1}(e) + \overline{b}\varphi_{i-1} + \sum_{e\in P_i} c(e)\cdot\left(\frac{\epsilon c_i}{c(e)}\cdot\ell_{i-1}(e)\right) + \overline{b}\varphi_{i-1}\cdot\frac{\epsilon c_i\cdot b(P_i)}{\overline{b}} \\ &= D(i-1) + \epsilon c_i(\ell_{i-1}(P_i) + \varphi_{i-1}b(P_i)). \end{aligned}$$

Since P_i is a $(1+\epsilon)$-approximate shortest path with respect to $\alpha(i-1)$, we get that:

$$D(i) \leq D(i-1) + \epsilon(1+\epsilon)c_i\alpha(i-1).$$

Let $\beta = \min_{\ell,\varphi} D(\ell, \varphi)/\alpha(\ell, \varphi)$ be the optimal value of the dual LP $\boldsymbol{D}_{LP}$. Then:

$$\begin{aligned} D(i) &\leq D(i-1) + \epsilon(1+\epsilon)c_i D(i-1)/\beta \\ &\leq D(i-1)\cdot e^{\epsilon(1+\epsilon)c_i/\beta}. \end{aligned}$$

If t is the index of the last iteration, then $D(t) \geq 1$. Since $D(0) \leq 2\delta m$:

$$1 \leq D(t) \leq 2\delta m\cdot e^{\epsilon(1+\epsilon)\sum_{i=1}^{t} c_i/\beta}.$$

Taking a ln from both sides, we get:

$$0 \leq \ln(2\delta m) + \epsilon(1+\epsilon)\sum_{i=1}^{t} c_i/\beta. \tag{4}$$

Let $F = \sum_{i=1}^{t} c_i$. Note that F is exactly the total amount of flow by which we augment over all iterations. Therefore, inequality (4) can be rewritten as:

$$\frac{F\epsilon(1+\epsilon)}{\beta} \geq \ln(1/(2\delta m)).$$

From Lemma C.4, since the scaled-down flow is a feasible solution for $\boldsymbol{P}_{LP}$, $F/\log_{1+\epsilon}(\frac{1+\epsilon}{\delta}) \leq \beta$ must hold. It remains to show that $F/\log_{1+\epsilon}(\frac{1+\epsilon}{\delta}) \geq (1-O(\epsilon))\beta$:

$$\begin{aligned}
\frac{F/\log_{1+\epsilon}(\frac{1+\epsilon}{\delta})}{\beta} &\geq \frac{\ln(1/(2\delta m))}{\epsilon(1+\epsilon)} \cdot \frac{1}{\log_{1+\epsilon}(\frac{1+\epsilon}{\delta})} \\
&= \frac{\ln(1/\delta) - \ln(2m)}{\epsilon(1+\epsilon)} \cdot \frac{\ln(1+\epsilon)}{\ln(\frac{1+\epsilon}{\delta})} \\
&\geq \frac{(1-\epsilon)\ln(1/\delta)}{\epsilon(1+\epsilon)} \cdot \frac{\ln(1+\epsilon)}{\ln(\frac{1+\epsilon}{\delta})} \\
&\geq 1 - O(\epsilon),
\end{aligned}$$

where the penultimate inequality uses the fact that $\delta = (2m)^{-1/\epsilon}$, so $2m = (1/\delta)^{\epsilon}$, and $\ln(2m) = \epsilon\ln(1/\delta)$, and the last inequality holds because $\ln(1+\epsilon) \geq \epsilon - \epsilon^2/2$ and $\ln(\frac{1+\epsilon}{\delta}) \leq (1+\epsilon)\ln(1/\delta)$.

□

C.2 Efficient Implementation Using Decremental SSSP

In this section, we complete the proofs of Theorem C.1 and Theorem C.2 by providing an efficient implementation of Algorithm 4 from Appendix C.1. The algorithm exploits the algorithm for decremental SSSP from Theorem 1.1, that we denote by $\mathcal{A}$. A similar technique was used in [Mad10] and in [CK19]. Our proofs are almost the same as the ones in [CK19], except that we need to take care of the cost function b.

C.2.1 Simple Graphs with Unit Edge Capacities

We start with the proof of Theorem C.1. Let $G = (V, E)$ be the input undirected simple graph with n nodes and m edges. We assume that all edge capacities are unit. Let $b \in \mathbb{R}^E_{>0}$ be the edge costs, with $b_{\max} = \max_e b(e)$, $b_{\min} = \min_e b(e)$, and $B = b_{\max}/b_{\min}$. Let $\bar{b}$ be the cost bound. Let $\delta = (2m)^{-1/\epsilon}$ be the same as in Algorithm 4. For every edge $e \in E$, we let its weight be $w(e) = \ell(e) + \varphi b(e)$. We run Algorithm 4, but we will employ the algorithm $\mathcal{A}$ in order to compute $(1+\epsilon)$-approximate shortest s-t paths in G, with respect to the edge weights $w(e)$. In order to do so, we construct another simple undirected graph $G' = (V', E')$ as follows. Let $K = \log_{(1+\epsilon/3)} \frac{1+\epsilon}{\delta} = O\left(\frac{\log m}{\epsilon^2}\right)$. Recall that, at the beginning of the algorithm, for every edge $e \in E$, we set $\ell(e) = \delta$, and we set $\varphi = \delta/\bar{b}$. Therefore, the initial weight of edge e is $w(e) = \ell(e) + \varphi b(e) = \delta(1 + b(e)/\bar{b})$. As long as the algorithm does

not terminate, $\ell(e) < 1$ and $\varphi < 1/\bar{b}$ must hold, so $w(e) < 1 + b(e)/\bar{b}$. Therefore, over the course of the algorithm, $w(e)$ may grow from $\delta(1 + b(e)/\bar{b})$ to at most $(1 + b(e)/\bar{b})$. The idea is to discretize all possible values that edge $w(e)$ may attain by powers of $(1 + \epsilon/3)$.

We now define the new graph $G' = (V', E')$. For every vertex $v \in V$, we add $(K+1)$ vertices $v_0, \ldots, v_K$ to V'. For every edge $e = (u, v) \in E$, we add $(K+1)$ edges $e_0, \ldots, e_K$ to E', where for each $0 \leq i \leq K$, $e_i = (u_i, v_i)$, and the weight $w'(e_i) = \delta(1 + b(e)/\bar{b})(1 + \epsilon/3)^i$. Additionally, for each original vertex $v \in V$ and index $i \in \{1, \ldots, K\}$, we add an edge (v_0, v_i) of weight $w'(v_0, v_i) = 0$ to E'. Note that $|V'| = O(nK) = O\left(\frac{n \log m}{\epsilon^2}\right)$.

We run the algorithm $\mathcal{A}$ from Theorem 1.1 on graph G', where the length of each edge e' is $w'(e')$, and the error parameter is $\epsilon/3$. Note that the ratio L of largest to smallest non-zero edge length is $\frac{(1+b_{\max}/\bar{b})}{\delta(1+b_{\min}/\bar{b})} \leq B/\delta$. Note that some edges of G' have length 0. However, as we show later, we will never ask a query between a pair of vertices that lie within distance 0 from each other, and so, using Remark C.3, we can use algorithm $\mathcal{A}$, except that we need to replace the $\log L$ factor in its running time by factor $\log(\frac{Ln}{\epsilon}) = \log(\frac{Bn}{\epsilon\delta}) = O(\log(Bn)/\epsilon^{O(1)})$. Therefore, the total update time of the algorithm $\mathcal{A}$ is $\widehat{O}((|V'|^2 \log B)/\epsilon^{O(1)}) = \widehat{O}((n^2 \log B)/\epsilon^{O(1)})$.

Next, we need to describe the sequence of edge deletions in graph G'. The edges are deleted according to the following rule. For every edge $e = (u, v) \in E$, we delete an edge $e_i = (u_i, v_i) \in E'$ when $w'(e_i)$ becomes smaller than $\ell(e) + \varphi b(e)$. These are the only edge deletions in G'. Lastly, we show that, given a $(1 + \epsilon/3)$-approximate shortest s_0-t_0 path in G' (with respect to edge lengths $w'(e')$), we can efficiently obtain a $(1 + \epsilon)$-approximate shortest s-t path in G (with respect to edge lengths $w(e)$).

Claim C.7 *At any time before Algorithm 4 terminates, given any $(1 + \epsilon/3)$-approximate w'-shortest s_0-t_0 path P' in G', we can construct, in time $O(|P'|)$, a $(1+\epsilon)$-approximate w-shortest s-t path P in G.*

Proof: Since we assume that Algorithm 4 did not yet terminate, $\sum_{e \in E} \ell(e) + \bar{b}\varphi < 1$, and so for every edge $e \in E$, $\delta(1 + b(e)/\bar{b}) \leq \ell(e) + \varphi b(e) \leq 1 + b(e)/\bar{b}$. From the definition of the edge deletion sequence, if i' is the smallest index for which the edge $e_{i'}$ lies in E', then $\ell(e) + \varphi b(e) \leq w(e_{i'}) < (\ell(e) + \varphi b(e))(1 + \epsilon/3)$.

Let dist denote the distance from s to t in G with respect to edge lengths $w(e)$, and let dist' denote the distance from s_0 to t_0 with respect to edge lengths $w'(e')$. Then $\mathsf{dist} \leq \mathsf{dist}' < \mathsf{dist} \cdot (1 + \epsilon/3)$ must hold.

Assume now that we are given a $(1+\epsilon/3)$-approximate shortest s_0-t_0 in G' with respect to edge lengths $w'(e')$. Then, by contracting every subpath $(v_i, v_0, v_j) \subseteq P$ of length 0 which corresponds to the same node v in G, we obtain an s-t path P in G whose length is $w(P') \leq (1 + \epsilon/3)\mathsf{dist}' < (1 + \epsilon/3)^2\mathsf{dist} \leq (1 + \epsilon)\mathsf{dist}$. □

Our algorithm only employs query $\mathsf{path\text{-}query}$ for the vertex t_0. In particular, it is easy to see that the distance from s_0 to t_0 is always non-zero. Therefore, we obtain a correct implementation of Algorithm 4. We now analyze its running time. As already observed, the total running time needed to maintain the data structure from Theorem 1.1 is $\widehat{O}((n^2 \log B)/\epsilon^{O(1)})$. Observe that in every iteration of Algorithm 4, we employ a single call to $\mathsf{path\text{-}query}(t_0)$ in graph G'. Each such query takes $\widehat{O}(|P| \log\log(Ln/\epsilon)) = \widehat{O}(n \log\log(B/\epsilon))$ time to return a path P and, by Lemma C.5, the number of queries is bounded by $\tilde{O}(n/\epsilon^2)$. Therefore, the total time needed to respond to all queries is bounded by $\widehat{O}\left((n^2 \log B)/\epsilon^{O(1)}\right)$. The running time of other steps for implementing Algorithm 4, such as maintaining ℓ and φ, are subsumed by these bounds. Altogether, the total running time is $\widehat{O}(n^2 \cdot \frac{\log B}{\epsilon^{O(1)}})$.

C.2.2 Vertex-Capacitated Graphs

We now complete the proof Theorem C.2. Our proof is almost identical to that of [CK19]. Let $G = (V, E)$ be the input undirected simple graph with n nodes and m edges. Let $b \in \mathbb{R}^V_{>0}$ be the vertex costs, with $b_{\max} = \max_v b(v)$, $b_{\min} = \min_v b(v)$, and $B = b_{\max}/b_{\min}$. Let $\bar{b}$ be the cost bound. Additionally, let $c \in \mathbb{R}^V_{>0}$ be the vertex capacities, with $c_{\max} = \max_v c(v)$, $c_{\min} = \min_v c(v)$, and $C = c_{\max}/c_{\min}$.

We proceed as follows. First, we use a standard reduction from vertex-capacitated flow problems in undirected graphs to edge-capacitated flow problems in directed graphs, constructing a directed graph G'' with capacities on edges. We will run Algorithm 4 on G''. In order to compute approximate $(\ell+\varphi b)$-shortest s-t paths in G'', we will employ the algorithm $\mathcal{A}$ for decremental SSSP from Theorem 1.1 in another graph G' – a simple undirected edge-weighted graph that we will construct. We now describe the construction of both graphs.

Graph G''. We construct a directed graph $G'' = (V'', E'')$ with edge capacities $c''(e)$ and edge costs $b''(e)$ for $e \in E''$, using a standard reduction from the input graph $G = (V, E)$. The set V'' of vertices contains, for every vertex $v \in V$ of the original graph, a pair v_{in}, v_{out} of vertices. Additionally, we add a directed edge (v_{in}, v_{out}) of capacity $c''(v_{in}, v_{out}) = c(v)$ and cost $b''(v_{in}, v_{out}) = b(v)$ to E''. For each undirected edge $(u, v) \in E$, we add a pair of new edges $(v_{out}, u_{in}), (u_{out}, v_{in})$ to E'', both with capacity ∞ and cost 0. This completes the definition of the graph G'', that we view as a flow network with source s_{out} and destination t_{in}. Observe that for any s-t flow f in G, there is a corresponding s_{out}-t_{in} flow f'' in G'', of the same value and cost, such that f is capacity-feasible iff f'' is capacity-feasible. Similarly, any s_{out}-t_{in} flow f'' in G'' can be transformed, in time $O(m)$, into an s-t flow of the same value and the same cost in G, such that f is capacity-feasible iff f'' is capacity-feasible. We run Algorithm 4 on G'', and maintain edge lengths $\ell'' \in \mathbb{R}^{E''}_{\geq 0}$ and a value $\varphi \geq 0$. It now remains to show how we compute a $(1+\epsilon)$-approximate $(\ell'' + \varphi b'')$-shortest s_{out}-t_{in} path in graph G''. In order to do so, we define a new graph G', on which we will run the algorithm $\mathcal{A}$ for decremental SSSP.

As before, for every edge $e \in E''$, we maintain a weight $w''(e) = \ell''(e) + \varphi b''(e)$. Recall that, at the beginning of the algorithm, we set $\ell''(e) = \delta/c''(e)$ if $c''(e)$ is finite, and we set $\ell''(e) = 0$ otherwise. We also set $\varphi = \delta/\bar{b}$. Therefore, initially, $w''(e) = \delta(1/c''(e) + b''(e)/\bar{b})$ (if $c''(e) = \infty$, then $w''(e) = \delta b''(e)/\bar{b} = 0$, and it remains 0 throughout the algorithm). As long as the algorithm does not terminate, $\ell''(e) < 1/c''(e)$ and $\varphi < 1/\bar{b}$ must hold, so $w''(e) < (1/c''(e) + b''(e)/\bar{b})$. Therefore, over the course of the algorithm, $w''(e)$ may increase from $\delta(1/c''(e) + b''(e)/\bar{b})$ to $(1/c''(e) + b''(e)/\bar{b})$.

Graph G'. We construct an undirected simple graph $G' = (V', E')$, from the original input graph $G = (V, E)$. We first place weights on the vertices of G', and later turn it into an edge-weighted graph. As before, we let $K = \log_{(1+\epsilon/3)} \frac{1+\epsilon}{\delta} = O\left(\frac{\log m}{\epsilon^2}\right)$. For every vertex $v \in V - \{s, t\}$, we add $K+1$ new vertices $v_0, \ldots, v_K$ to V', and for all $0 \leq i \leq K$, we set the weight $w(v_i) = \delta\left(\frac{1}{c(v)} + \frac{b(v)}{\bar{b}}\right) \cdot (1+\epsilon/3)^i$. For each edge $e = (u, v) \in E$ in the original graph, we add $(K+1)^2$ new edges $e_{i,j} = (u_i, v_j)$ for all $i, j \in \{0, \ldots, K\}$ to E'. We also add two new vertices s and t to V', with weight 0. For each edge $e = (s, u) \in E$, for all $0 \leq i \leq K$, we add an edge $e^s_i = (s, u_i)$ to E'. Similarly, for each edge $e = (u, t) \in E$, for all $0 \leq i \leq K$, we add an edge $e^t_i = (u_i, t)$ to E'. Observe that $|V'| = O(nK) = O\left(\frac{n \log n}{\epsilon^2}\right)$.

We would like to run the algorithm $\mathcal{A}$ for decremental SSSP on the graph G'. However, G' has weights on vertices and not edges. This can be easily fixed as follows. For each edge $e = (u, v)$ in G', we let its weight be $w(e) = (w(u) + w(v))/2$. Since $w(s) = w(t) = 0$, for every s-t path P' in G', the total

weight of all edges on P' equals to the total weight of all vertices on P'. Note that all edge weights are non-zero.

We run the algorithm $\mathcal{A}$ from Theorem 1.1 on graph G', where the length of each edge e is $w(e)$, and the error parameter is $\epsilon/3$.

Notice that the ratio L of largest to smallest edge length in G' is $L = \frac{1/c_{\min}+b_{\max}/\bar{b}}{\delta(1/c_{\max}+b_{\min}/\bar{b})} \leq \frac{CB}{\delta}$. By Theorem 1.1, the total running time of $\mathcal{A}$ is $\widehat{O}\left(\frac{(nK)^2\log(L)}{\epsilon^2}\right) = \widehat{O}\left(n^2 \cdot \frac{\log(CB)}{\epsilon^{O(1)}}\right)$.

Next, we need to describe the sequence of edge deletions from the graph G'. The edges are deleted according to the following rule. For every vertex $v \in V$ in the original graph, for every $0 \leq i \leq K$, whenever $w''(v_{in}, v_{out}) > w(v_i)$, we delete all edges incident to v_i from G'. For convenience, we say that vertex v_i becomes *eliminated*. We use the following analogue of Claim C.7.

Claim C.8 *Throughout the execution of Algorithm 4, given any $(1+\epsilon/3)$-approximate s-t path P' in G' with respect to edge lengths $w(e)$, we can construct, in time $O(|P'|)$, a $(1+\epsilon)$-approximate s_{out}-t_{in} path P'', with respect to edge lengths $w''(e)$, in G''.*

Proof: Let P'' be the shortest s_{out}–t_{in} path in graph G'', with respect to the edge lengths w'', and assume that $P'' = (s_{out}, v^1_{in}, v^1_{out}, \ldots, v^z_{in}, v^z_{out}, t_{in})$. Let W'' denote the length of the path P''. For all $1 \leq j \leq z$, let $e^j = (v^j_{in}, v^j_{out})$. Since Algorithm 4 did not yet terminate, $w''(e^j) < 1/c''(e^j) + b''(e^j)/\bar{b}$. Therefore, if we let i_j be the smallest index, such that vertex $v^j_{i_j}$ of G' is not yet eliminated, then $w(v^j_{i_j}) \leq w''(e^j)(1+\epsilon/3)$. Consider now the following path in graph G': $P' = (s, v^1_{j_1}, v^2_{j_2}, \ldots, v^z_{j_z})$. Since no vertex on this path is eliminated, the path is indeed still contained in G'. The total weight of the vertices on this path is bounded by $(1+\epsilon/3)W''$. From the above discussion, the total w'-weight of the edges on this path is then also bounded by $(1+\epsilon/3)W''$.

We denote by dist'' the distance from s_{out} to t_{in} in G'', with respect to the edge lengths $w''(e)$, and we denote by dist' the distance from s to t in graph G', with respect to edge lengths $w(e)$. From the above discussion, $\mathsf{dist}' \leq (1+\epsilon/3)\mathsf{dist}''$.

Consider now a $(1+\epsilon/3)$-approximate s-t path P' in graph G', with respect to the edge lengths $w'(e)$, so the total weight $w'(e)$ of all edges on P' is at most $(1+\epsilon/3)\mathsf{dist}' \leq (1+\epsilon/3)^2\mathsf{dist}'' \leq (1+\epsilon)\mathsf{dist}''$. Assume that $P' = (s, v^1_{j_1}, v^2_{j_2}, \ldots, v^z_{j_z})$. Consider the following path in graph G'': $P = (s_{out}, v^1_{in}, v^1_{out}, \ldots, v^z_{in}, v^z_{out}, t_{in})$. Note that for all $1 \leq j \leq z$, the weight $w''(v^j_{in}, v^j_{out}) \leq w'(v^j)$ must hold (or vertex v^j would have been eliminated). Therefore, the total weight $w''(e)$ of the edges of P'' is bounded by the total weight $w(v)$ of the vertices of P', which in turn is equal to the total weight $w(e')$ of the edges of P', that is bounded by $(1+\epsilon)\mathsf{dist}''$. □

From the above claim, in every iteration of Algorithm 4, we can use $\mathsf{path\text{-}query}(t)$ in graph G' in order to compute the $(1+\epsilon)$-approximate shortest s_{out}-t_{in} path in G'', with respect to edge lengths $w'' = \ell'' + \varphi b''$. It now remains to analyze the running time of the algorithm. Each query to the decremental SSSP data structure takes time $\widehat{O}(|P|\log\log L) = \widehat{O}((n\log(BC)/\epsilon^{O(1)})$ to return a path P and, from Lemma C.5(2), there are at most $\tilde{O}(n/\epsilon^2)$ such queries. Therefore, the total time spent on responding to the queries is $\widehat{O}\left(\frac{n^2\log(BC)}{\epsilon^{O(1)}}\right)$. As observed above, the total expected running time for maintaining the decremental SSSP data structure is $\widehat{O}\left(n^2 \cdot \frac{\log(CB)}{\epsilon^{O(1)}}\right)$. The running time of other steps for implementing Algorithm 4, such as maintaining ℓ'' and φ, is subsumed by these running times. Overall, the total running time is $\widehat{O}\left(n^2 \cdot \frac{\log(CB)}{\epsilon^{O(1)}}\right)$.

D Additional Applications

In this section, we describe additional applications of decremental SSSP, and some new results that follow from our algorithm from Theorem 1.1, as well as additional results that could be obtained from the algorithm of [CK19].

D.1 Concurrent k-commodity Bounded-Cost Flow

In the concurrent k-commodity bounded-cost flow problem, we are given a graph G with capacities and costs on either edges or nodes, and a cost bound $\bar{b}$. We are also given k *demands* represented by tuples $(s_1, t_1, d_1), \ldots, (s_k, t_k, d_k)$, where for all $1 \leq i \leq k$, s_i and t_i are vertices of G, that we refer to as a *demand pair*, and d_i is a non-negative real number. The goal is to find a largest value $\lambda > 0$, and to compute, for all $i \in \{1, \ldots, k\}$, an s_i-t_i flow f_i of value λd_i (that is, flow f_i routes λd_i units of flow from s_i to t_i). We say that the resulting flow $f = \bigcup_i f_i$ is edge-capacity-feasible, iff for all $e \in E$, $\sum_{i=1}^{k} f_i(e) \leq c(e)$. We say that the flow f is edge-cost-feasible, if $\sum_{e \in E}(\sum_{i \leq k} f_i(e))b(e) \leq \bar{b}$. The goal is to maximize λ, while ensuring that the resulting flow f is both capacity-feasible and cost-feasible. The problem with vertex capacities and cost is defined analogously.

The concurrent k-commodity flow problem is defined in the same way, except that we no longer have costs on edges or vertices, and we do not require that the flow is cost-feasible.

We denote by $T_{\mathsf{MBCF}}(n, m, \epsilon, B, C)$ the time needed for computing a $(1+\epsilon)$-approximate solution for an MBCF problem instance, in a graph with n nodes and m edges, where B is the ratio of largest to smallest (edge or vertex) costs, C is the ratio of largest to smallest (edge or vertex) capacities. We use the following result.

Lemma D.1 (Concurrent k-commodity bounded-cost flow [GK96, Fle00]) *There is an algorithm that, given a graph G with n nodes, m edges, (edge or vertex) capacities c, where C is the ratio of largest to smallest capacity, (edge or vertex) costs b, and B is the ratio of largest to smallest cost, and a set of k demands, computes a $(1+\epsilon)$-approximate concurrent k-commodity bounded-cost flow in time $\tilde{O}\left(\frac{k}{\epsilon^2} \cdot T_{\mathsf{MBCF}}(n, m, \epsilon, BC/\delta, C) \cdot \log(BC)\right)$ where $\delta = (2m)^{-1/\epsilon}$.*

Proof: [Sketch] A similar lemma was shown by Garg and Könemann in Section 6 of [GK96]. However, the term $T_{\mathsf{MBCF}}(n, m, \epsilon, BC/\delta, C)$ in [GK96] was the time for computing *exact* min-cost flow. We sketch here why only $(1+\epsilon)$-approximate solution for MBCF is sufficient.

For any commodity $1 \leq i \leq k$ and edge lengths $\ell \in \mathbb{R}^E_{\geq 0}$, let $\texttt{mincost}_i(\ell)$ be minimum cost for sending a flow of d_i units from s_i to t_i in $G = (V, E, c)$, where the edge costs are defined by ℓ. It was shown in [GK96], that, in order to solve concurrent k-commodity bounded-cost flow, it is enough to solve the following problem $O(\frac{1}{\epsilon^2} k \log k \log m)$ times: given i and ℓ, compute an s_i-t_i flow $f_{i,\ell}$ of value d_i and cost $\texttt{mincost}_i(\ell)$ w.r.t. ℓ.

Let $\mathcal{A}$ be the $(1+\epsilon)$-approximate algorithm for MBCF. We claim that, by calling this algorithm $O(\log BC)$ times, we can compute an s_i-t_i flow $f'_{i,\ell}$ such that the value of $f'_{i,\ell}$ is exactly $d_i/(1+\epsilon)$, and its cost is at most $\texttt{mincost}_i(\ell)$. Indeed, observe that, when given $\texttt{mincost}_i(\ell)$ as a cost bound to $\mathcal{A}$, algorithm $\mathcal{A}$ will return a flow of value at least $d_i/(1+\epsilon)$. By scaling, we obtain a flow of value exactly $d_i/(1+\epsilon)$ and cost at most $\texttt{mincost}_i(\ell)$.

By following the analysis of [GK96], it is easy to see that that we can use $f'_{i,\ell}$ instead of $f_{i,\ell}$, for any given i and ℓ. Every step in the analysis works as it is except that the size of the solution at the end is reduced by factor $(1+\epsilon)$.

□

By plugging Theorem C.1 and Theorem C.2 into the above lemma, we obtain the following corollary:

Corollary D.2 *There is a deterministic algorithm for computing a $(1+\epsilon)$-approximate concurrent k-commodity bounded-cost flow in time $\widehat{O}(kn^2\frac{\log^2 BC}{\epsilon^{O(1)}})$ on either:*

- *undirected simple graphs with unit edge-capacities and arbitrary edge-costs; or*
- *undirected simple graphs with arbitrary vertex-capacities and vertex-costs.*

Our algorithm for concurrent k-commodity flow is slower than the $\tilde{O}(mk)$ algorithm of Sherman [She17]. However, in the bounded-cost version, our algorithms are faster than the previous best algorithms whenever $m = \omega(n^{1.5+o(1)})$ and $k = O((m/n)^2)$; see Table 3. We note that the algorithm for vertex-capacitated graphs can also be obtained from the results of [CK19], except that the resulting algorithm would be randomized.

D.2 Maximum k-commodity Bounded-Cost Flow

In the maximum k-commodity bounded-cost flow, we are given a graph G with capacities and costs on either edges or nodes, and a cost bound $\bar{b}$. We are also given k *demand pairs* $(s_1,t_1),\ldots,(s_k,t_k)$. The goal is to compute, for all $i \in \{1,\ldots,k\}$, an s_i-t_i flow f_i with the following constraints. Let $f = \bigcup_i f_i$ be the resulting k-commodity flow. We say that the flow is edge-capacity-feasible, if $\sum_{i=1}^k f_i(e) \le c(e)$ for all $e \in E$, and we say that it is edge-cost-feasible, if $\sum_{e\in E}(\sum_{i=1}^k f_i(e))b(e) \le \bar{b}$. Let $|f_i|$ denote the value of the flow f_i – the amount of flow sent from s_i to t_i. The goal is to find the flows $f_1,\ldots,f_k$ that maximize $\sum_i |f_i|$, such that the resulting flow $f = \bigcup_i f_i$ is both edge-capacity-feasible and edge-cost feasible. The problem with vertex capacities and cost is defined similarly.

The maximum k-commodity flow problem is defined similarly, except that there are no costs on edges or vertices, and we do not require that the flow f is cost-feasible.

We obtain the following corollary.

Corollary D.3 *There is a deterministic algorithm, that, given a graph G with n nodes, m edges, capacities c where C is the ratio of largest to smallest capacity, costs b where B is the ratio of largest to smallest cost, and a set of k demand pairs, can compute a $(1+\epsilon)$-approximate maximum k-commodity bounded-cost flow in time $\widehat{O}(kn^2\frac{\log BC}{\epsilon^{O(1)}})$ on either:*

- *undirected simple graphs with unit edge-capacities and arbitrary edge-costs; or*
- *undirected simple graphs with arbitrary vertex-capacities and vertex-costs.*

As before, the result for vertex-capacitated graphs could also be obtained from [CK19], except that the resulting algorithm would be randomized. To our best knowledge, unlike the concurrent k-commodity flow, there is no black-box reduction from maximum k-commodity flow or maximum k-commodity bounded-cost flow to MBCF. However, Corollary D.3 can be proved by extending the MWU-based technique used in Appendix C to the maximum k-commodity bounded-cost flow, and employing the algorithm for dynamic SSSP for executing each iteration efficiently; we omit the proof.

Our algorithm for maximum k-commodity flow is faster than the $O(k^{O(1)}m^{4/3}/\epsilon^{O(1)})$-time algorithm by [KMP12] and the $\tilde{O}(m^2/\epsilon^2)$-time algorithm by [Fle00] whenever $m = \omega(n^{1.5+o(1)})$ and $k = O((m/n)^2)$.

The same bounds hold in the bounded-cost version and in the vertex-capacitated setting: our algorithms are faster than the previous best algorithms whenever $m = \omega(n^{1.5+o(1)})$ and $k = O((m/n)^2)$; see Table 3.

Lastly, we describe several additional applications of the SSSP problem, that can be obtained from the algorithm of [CK19] (as well as from our algorithm from Theorem 1.1)

D.3 Most-Balanced Sparsest Vertex Cut

Given a graph $G = (V, E)$, a vertex cut is a partition (A, B, C) of the vertex set V, such that there are no edges between A and C, and $A, C \neq \emptyset$. The sparsity of the cut (A, B, C) is $h_G(A, B, C) = \frac{|B|}{\min\{|A|,|C|\}+|B|}$. We say that a cut (A, B, C) is φ-sparse if $h_G(A, B, C) < \varphi$. The *most balanced* φ-sparse cut is a φ-sparse cut (A, B, C) such that $\min\{|A|, |C|\}$ is maximized. The vertex expansion of a graph G is $h_G = \min\{h_G(A, B, C) \mid (A, B, C) \text{ is a vertex cut of } G\}$.

In the α-approximate most-balanced sparsest vertex cut problem, we are given a graph $G = (V, E)$ and a parameter h_G. Let (A', B', C') be a most-balanced h_G-sparsest cut. The goal is to find a vertex cut (A, B, C) with $h_G(A, B, C) \leq \alpha \cdot h_G(A', B', C')$, such that $\min\{|A|, |C|\} \geq \min\{|A'|, |C'|\}/3$. The following result follows from the algorithm of [CK19].

Lemma D.4 (Most-balanced sparsest vertex cut) *There is a randomized algorithm, that, given a graph G with n nodes, computes a $O(\log^2 n)$-approximate most-balanced sparsest vertex cut in time $O(T_{mf}(n, 2, n) \log^2 n)$ where and $T_{mf}(n, \epsilon, C)$ is the time required to compute a $(1 + \epsilon)$-approximate maximum s-t flow and a $(1 + \epsilon)$-approximate minimum s-t cut in an n-vertex graph with vertex capacities, where C is the ratio of largest to smallest vertex capacity.*

The lemma follows from the cut-matching game framework of Khandekar, Rao, and Vazirani [KRV09]. The algorithm of [KRV09] is designed to compute a sparsest cut or minimum balanced cut in edge-capacitated graphs, but this is only because it relies on maximum flow / minimum cut computation in edge-capacitated graphs. By computing approximate maximum flow / minimum cut in vertex-capacitated graphs, one can immediately obtain Lemma D.4. By plugging Theorem C.2 into the above lemma, we obtain the following corollary:

Corollary D.5 *There is a randomized algorithm for computing a $O(\log^2 n)$-approximate most-balanced sparsest vertex cut in a given n-vertex graph G, in expected time $\widehat{O}(n^2)$.*

D.4 Treewidth and Tree Decompositions

Given a graph $G = (V, E)$, a tree decomposition of G consists of a tree T, and, for every vertex $a \in V(T)$, a subset $X_a \subseteq V$ of vertices of G, that satisfy the following conditions: (i) for each edge $(u, v) \in E$ of G, there is a tree-node $a \in V(T)$ with $u, v \in X_a$; and (ii) for each vertex $u \in V$ of G, all tree-nodes $a \in V(T)$ with $u \in X_a$ induce a non-empty connected subgraph of T. The width of the tree decomposition is $\max_{a \in V(T)} |X_a| - 1$. The treewidth of G is the minimum width of a tree decomposition of G. Treewidth and tree decomposition are used extensively in algorithmic graph theory and in fixed parameter tractable (FPT) algorithms.

The following lemma reduces the problem of approximating treewidth to the most-balanced sparsest vertex cut problem, using standard techniques. We omit the proof; see also [BGHK95].

Lemma D.6 *There is an algorithm that, given an n-vertex graph G and a parameter α, computes a tree decomposition of G of width $O(k\alpha \log n)$, where k is the treewidth of G, in time $\tilde{O}(T_{svc}(n, \alpha))$*

where $T_{svc}(n, \alpha)$ is the time needed for computing an α-approximate most-balanced sparsest vertex cut in G.

By plugging Corollary D.5 into the above lemma, we obtain the following corollary:

Corollary D.7 *There is a deterministic algorithm that, given an n-vertex graph G, computes a tree decomposition of G, of width $O(k \log^3 n)$, where k is the treewidth of G, in time $\widehat{O}(n^2)$.*

For comparison, given a graph with n nodes and treewidth k, previous algorithms either have running time exponentially depending on k [RS95, Ami01, Ami10, Bod96, BDD+16] or have a large polynomial running time [BGHK95, Ami01, Ami10, FHL08]. One exception is the algorithm by Fomin et al. [FLS+18] which computes an $O(k)$-approximation of treewidth in time $O(k^7 n \log n)$; see Table 5 for a summary. Our algorithm is faster than [FLS+18] when $k \geq n^{1/7+o(1)}$ and also gives a better approximation.

Although most of fixed parameter tractable (FPT) algorithms only concern graphs with constant treewidth $k = O(1)$ or very small k, there is a recent line of work on *fully-polynomial* FPT algorithms [FLS+18, IOO18] for many fundamental graph problems including maximum matching and max flow, and matrix problems including determinant and solving linear system, which concern instances whose treewidth can be polynomial. In those settings, the approximation factor of $O(\log^3 n)$ from Corollary D.7 is of interest.

E Tables

	Year	(α, β)-approximation	Total update time	Query time for dist-query	Weighted?	Directed?	Det?
[ES81]*	1981	$(1, 0)$	$O(mn^2)$	$O(1)$		Directed	Det
[BHS07]*	2002	$(1, 0)$	$\tilde{O}(n^3)$	$O(1)$		Directed	
[BHS07]	2002	$(1+\epsilon, 0)$	$\tilde{O}(n^2\sqrt{m}/\epsilon)$	$O(1)$		Directed	
[RZ12]	2004	$(1+\epsilon, 0)$	$\tilde{O}(mn/\epsilon)$	$O(1)$			
[FHN16]*	2013	$(1+\epsilon, 0)$	$\tilde{O}(mn/\epsilon)$	$O(\log\log n)$			Det
[Ber16]*	2013	$(1+\epsilon, 0)$	$\tilde{O}(mn\log L/\epsilon)$	$O(1)$	Weighted	Directed	
[RZ11, FHNS15]	2004	(α, β): $2\alpha+\beta<4$	$\Omega(n^{3-o(1)})$	$\Omega(n^{1-o(1)})$			
[BKS12] (cf. [FG19])*	2008	$(2k-1, 0)$	$\tilde{O}(m)$	$\tilde{O}(n^{1+1/k})$	Weighted		
[BvdBG+20]*	2020	$(\text{poly}\log n, 0)$	$\tilde{O}(m)$	$\tilde{O}(n)$	Weighted		Random adaptive
[BR11]	2011	$(2k-1+\epsilon, 0)$	$O(n^{2+1/k+o(1)})$	$O(k)$			
[FHN16]*	2013	$(2+\epsilon, 0)$ or $(1+\epsilon, 2)$	$\tilde{O}(n^{2.5}/\epsilon)$	$O(1)$			
[ACT14]	2014	$(2^{O(k\rho)}, 0)$	$O(mn^{1/k})$	$O(k\rho)$			
[FHN14a]	2014	$((2+\epsilon)^k-1, 0)$	$O(m^{1+1/k+o(1)}\log^2 L)$	$O(k^k)$	Weighted		
[Che18]*	2018	$((2+\epsilon)k-1, 0)$	$O(mn^{1/k+o(1)}\log L)$	$O(\log\log(nL))$	Weighted		
This paper*		$(3\cdot 2^k, \gamma^{O(k)})$	$\widehat{O}(n^{2.5+2/k}\gamma^{O(k)})$	$O(\log\log n)$			Det

Table 1: Upper and lower bounds for decremental APSP. We denote by n the number of graph vertices, by m the initial number edges, L is the ratio of largest to smallest edge length, and k is a positive integral parameter. We also use parameters $\rho = (1 + \left\lceil \frac{\log n^{1-1/k}}{\log(m/n^{1-1/k})} \right\rceil) \leq k$, $0 < \epsilon < 1$, and $\gamma = \exp(\log^{3/4} n)$. In the "Year" column, the year is according to the conference version of the paper. If the algorithm only works for unweighted graphs or only undirected graphs, then we left the columns "Weighted?" and "Directed?", respectively, blank. If the result assumes an oblivious adversary, then we left the column "Det?" blank. Otherwise, we write "Det" or "Random adaptive" which means that the result is deterministic or randomized but works against an adaptive adversary, respectively. The algorithms without the "*" mark are subsumed by other algorithms with the "*" mark, to within $n^{o(1)}$ factors. The fact that the algorithm in [ES81] can be extended to work in directed graphs was observed in [HK95]. The algorithms by [BKS12, FG19, BvdBG+20] are actually fully dynamic algorithms for maintaining spanners, but they automatically imply decremental APSP with large query time for dist-query.

	Year	Approx.	Total update time	Handles path-query?	Query time for path-query	Weighted?	Det?	Notes
[ES81]*	1981	exact	$O(mn)$	Yes	$O(\|P\|)$		Det	
[RZ11, FHNS15]	2004	exact	$\Omega(mn^{1-o(1)})$					
[BR11]	2011	$1+\epsilon$	$O(n^{2+o(1)})$	Yes	$O(\|P\|)$			
[FHN14b]	2014	$1+\epsilon$	$O(n^{1.8+o(1)} + m^{1+o(1)})$	Yes	$O(\|P\|)$			
[FHN14a]*	2014	$1+\epsilon$	$O(m^{1+o(1)} \log L)$	Yes	$\tilde{O}(\|P\|)$	Weighted		
[BC16]	2016	$1+\epsilon$	$\tilde{O}(n^2)$				Det	
[BC17]	2017	$1+\epsilon$	$\tilde{O}(n^{5/4}\sqrt{m})$				Det	
[Ber17]	2017	$1+\epsilon$	$\tilde{O}(n^2 \log L)$			Weighted	Det	
[CK19]	2019	$1+\epsilon$	$\widehat{O}(n^2 \log L)$	Yes	$\tilde{O}(n \log L)$	Weighted	Random adaptive	Vertex deletions only
[GWN20]	2020	$1+\epsilon$	$\widehat{O}(m\sqrt{n})$				Det	
[BvdBG+20]*	2020	$1+\epsilon$	$\widehat{O}(m\sqrt{n})$	Yes	$\tilde{O}(n)$		Random adaptive	
This paper*		$1+\epsilon$	$\widehat{O}(n^2 \log L)$	Yes	$\widehat{O}(\|P\|)$	Weighted	Det	

Table 2: Upper and lower bounds for decremental SSSP. We denote by n the number of graph vertices, by m the initial number of edges, L is the ratio of largest to smallest edge length, and $0 < \epsilon < 1$ is a given parameter. The dependency of the running time on ϵ is omitted for simplicity. We denote by P the (approximate) shortest path returned in response to path-query, and by $|P|$ the number of edges in P. In the "Year" column, the year is according to the conference version of the paper. If a result works only on unweighted graphs, then we left the column "Weighted?" blank. If a result assumes an oblivious adversary, then we left the column "Det?" blank. Otherwise, we write "Det" or "Random adaptive" which means that the result is deterministic or randomized but works against an adaptive adversary, respectively. The algorithms without the "*" mark are subsumed by other algorithms with the "*" mark, to within $\operatorname{poly}\log n$ factors.

Problems in unit edge-capacity setting	**Previous best**	**This paper**	**Faster when**
maximum s-t flow	$\tilde{O}(m/\epsilon)$ [She17]	$\widehat{O}(n^2/\epsilon^{O(1)})$	-
k-commodity concurrent flow	$\tilde{O}(km/\epsilon)$ [She17]	$\widehat{O}(kn^2/\epsilon^{O(1)})$	-
maximum k-commodity flow	$O(k^{O(1)}m^{4/3}/\epsilon^{O(1)})$ [KMP12] $\tilde{O}(mn/\epsilon^2)$ [Mad10]	$\widehat{O}(kn^2/\epsilon^{O(1)})$	$m = \omega(n^{1.5+o(1)})$ and $k = O((m/n)^2)$
maximum bounded-cost s-t flow	$\tilde{O}(m\sqrt{n})$ [LS14] (exact) $\tilde{O}(m^{10/7})$ [CMSV17](exact)	$\widehat{O}(n^2/\epsilon^{O(1)})$	$m = \omega(n^{1.5+o(1)})$
k-commodity concurrent bounded-cost flow	$\tilde{O}(km\sqrt{n}/\epsilon^{O(1)})$ [LS14]+Lemma D.1 $\tilde{O}(m(m+k)/\epsilon^2)$ [Fle00]	$\widehat{O}(kn^2/\epsilon^{O(1)})$	$m = \omega(n^{1.5+o(1)})$ and $k = O((m/n)^2)$
maximum k-commodity bounded-cost flow	$\tilde{O}(m(m+k)/\epsilon^2)$ [Fle00]	$\widehat{O}(kn^2/\epsilon^{O(1)})$	$k = O((m/n)^2)$

Table 3: Best currently known running times of algorithms for flow and cut problems in undirected simple graphs with unit edge capacities. We use the $\tilde{O}$ notation to hide factors that are polylogarithmic in n and B – the ratio of maximum to minimum edge cost. All algorithms obtain a $(1+\epsilon)$-approximation for the corresponding problem, unless explicitly stated otherwise.

Problem in vertex-capacitated setting	**Previous best**	**This paper / follows from [CK19]**	**Faster when**
maximum s-t flow	$\widehat{O}(n^2/\epsilon^{O(1)})$ [CK19] $\tilde{O}(m\sqrt{n})$ [LS14] (exact)	$\widehat{O}(n^2/\epsilon^{O(1)})$	-
k-commodity concurrent flow	$\tilde{O}(km\sqrt{n}/\epsilon^{O(1)})$ [LS14]+Lemma D.1, $\tilde{O}(mn/\epsilon^2)$ [Mad10]	$\widehat{O}(kn^2/\epsilon^{O(1)})$	$m = \omega(n^{1.5+o(1)})$ and $k = O((m/n)^2)$
maximum k-commodity flow	$\tilde{O}(mn/\epsilon^2)$ [Mad10]	$\widehat{O}(kn^2/\epsilon^{O(1)})$	$k = O((m/n)^2)$
maximum bounded-cost s-t flow	$\tilde{O}(m\sqrt{n})$ [LS14] (exact)	$\widehat{O}(n^2/\epsilon^{O(1)})$	$m = \omega(n^{1.5+o(1)})$
k-commodity concurrent bounded-cost flow	$\tilde{O}(km\sqrt{n}/\epsilon^{O(1)})$ [LS14]+Lemma D.1 $\tilde{O}(m(m+k)/\epsilon^2)$ [Fle00]	$\widehat{O}(kn^2/\epsilon^{O(1)})$	$m = \omega(n^{1.5+o(1)})$ and $k = O((m/n)^2)$
maximum k-commodity bounded-cost flow	$\tilde{O}(m(m+k)/\epsilon^2)$ [Fle00]	$\widehat{O}(kn^2/\epsilon^{O(1)})$	$k = O((m/n)^2)$
$O(\log^2 n)$-approximate sparsest cut	$\widehat{O}(n^2)$ [CK19] $\tilde{O}(m\sqrt{n})$ [LS14]+Lemma D.4	$\widehat{O}(n^2)$	-

Table 4: Best currently known algorithms for flow and cut problems in undirected graphs with vertex capacities. We use $\tilde{O}$ notation to hide factors that are polylogarithmic in n, C and B, where C is the ratio of maximum to minimum vertex capacity, and B is the ratio of maximum to minimum vertex cost. All algorithms are for obtaining a $(1+\epsilon)$-approximation for the corresponding problem, unless explicitly stated otherwise.

Reference	**Approximation**	**Time**
FPT time		
[RS95]	4	$2^{O(k)}n^2$
[Ami01, Ami10]	$3\frac{2}{3}$	$2^{O(k)}n^3$
[Bod96]	1	$k^{O(k^3)}n$
[BDD+16]	3	$2^{O(k)}n\log n$
	5	$2^{O(k)}n$
Polynomial time		
[BGHK95]	$O(\log n)$	$\text{poly}(n)$
[Ami01, Ami10]	$O(\log k)$	$k^5n^3\text{polylog}(nk)$
[FHL08]	$O(\sqrt{\log k})$	$\text{poly}(n)$
[FLS+18]	$O(k)$	k^7n
This paper / follows from [CK19]	$O(\log^3 n)$	$n^{2+o(1)}$

Table 5: Algorithms for approximating treewidth of a graph with n nodes and treewidth k.

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, 67

[AHdLT05] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. 33

[AHK12] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012. 54

[Ami01] Eyal Amir. Efficient approximation for triangulation of minimum treewidth. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, August 2-5, 2001*, pages 7–15, 2001. 66, 69

[Ami10] Eyal Amir. Approximation algorithms for treewidth. *Algorithmica*, 56(4):448–479, 2010. 66, 69

[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. 1, 2, 6, 68

[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. 1, 2, 68

[BDD+16] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A c^k n 5-approximation algorithm for treewidth. *SIAM J. Comput.*, 45(2):317–378, 2016. 66, 69

[Ber16] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM Journal on Computing*, 45(2):548–574, 2016. 2, 3, 67

[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. 1, 2, 3, 36, 38, 68

[BGHK95] Hans L. Bodlaender, John R. Gilbert, Hjálmtyr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. Algorithms*, 18(2):238–255, 1995. 65, 66, 69

[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, 11, 14

[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. 1

[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. 1

[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, 67

[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. 1

[BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4):35:1–35:51, 2012. 67

[Bod96] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996. 66, 69

[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. 1, 3, 4, 11, 67, 68

[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. 1, 2, 3, 4, 67, 68

[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. 1, 11, 13, 49, 50, 51, 52

[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, 11, 67

[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. 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 36, 38, 47, 52, 54, 56, 59, 61, 63, 64, 65, 68, 69

[CMSV17] Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\tilde{o}(m^{10/7}logw)$ time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 752–771, 2017. 55, 68

[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. 1

[DHZ00] Dorit Dor, Shay Halperin, and Uri Zwick. All-pairs almost shortest paths. *SIAM J. Comput.*, 29(5):1740–1759, 2000. 1, 3

[Din06] Yefim Dinitz. Dinitz' algorithm: The original version and Even's version. In *Theoretical computer science*, pages 218–240. Springer, 2006. 6

[ES81] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981. 2, 3, 6, 67, 68

[FG19] Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 377–388, 2019. 67

[FHL08] U. Feige, M.T. Hajiaghayi, and J.R. Lee. Improved approximation algorithms for minimum weight vertex separators. *SIAM Journal on Computing*, 38:629–657, 2008. 66, 69

[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. 1, 3, 4, 11, 67, 68

[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. 1, 68

[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. 1, 3, 4, 11, 40, 67

[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. 1, 2, 3, 67, 68

[Fle00] Lisa Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.*, 13(4):505–520, 2000. 54, 56, 63, 64, 68, 69

[FLS+18] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, Michal Pilipczuk, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Trans. Algorithms*, 14(3):34:1–34:45, 2018. 66, 69

[GK96] M. Goemans and J. Kleinberg. An improved approximation ratio for the minimum latency problem. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1996. 63

[GK98] Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 300–309, 1998. 54, 56

[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. 1, 2, 3, 11, 40, 68

[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. 3, 6, 33

[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, 1995. 6, 67

[IOO18] Yoichi Iwata, Tomoaki Ogasawara, and Naoto Ohsaka. On the power of tree-depth for fully polynomial FPT algorithms. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, pages 41:1–41:14, 2018. 66

[KKOV07] Rohit Khandekar, Subhash Khot, Lorenzo Orecchia, and Nisheeth K Vishnoi. On a cut-matching game for the sparsest cut problem. *Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2007-177*, 6(7):12, 2007. 50, 51

[KMP12] Jonathan A. Kelner, Gary L. Miller, and Richard Peng. Faster approximate multicommodity flow using quadratically coupled flows. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1–18, 2012. 64, 68

[KRV09] Rohit Khandekar, Satish Rao, and Umesh Vazirani. Graph partitioning using single commodity flows. *Journal of the ACM (JACM)*, 56(4):19, 2009. 50, 65

[KT19] Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *J. ACM*, 66(1):4:1–4:50, 2019. 19

[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, 56, 68, 69

[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. 1, 56, 59, 68, 69

[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. 1, 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. 1, 5

[RS95] Neil Robertson and Paul D Seymour. Graph minors. XIII. the disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995. 66, 69

[RZ11] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011. 1, 3, 67, 68

[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, 67

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

[She17] Jonah Sherman. Area-convexity, l_∞ regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 452–460, 2017. 64, 68

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. 1

[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, 10, 13, 48

[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. 1

[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. 1

[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