Decremental All-Pairs Shortest Paths in Deterministic Near-Linear Time[*]

Julia Chuzhoy[†]

September 10, 2021

Abstract

We study the decremental All-Pairs Shortest Paths (APSP) problem in undirected edge-weighted graphs. The input to the problem is an undirected n-vertex m-edge graph G with non-negative lengths on edges, that undergoes an online sequence of edge deletions. The goal is to support approximate shortest-paths queries: given a pair x, y of vertices of G, return a path P connecting x to y, whose length is within factor α of the length of the shortest x-y path, in time $\widetilde{O}(|E(P)|)$, where α is the approximation factor of the algorithm. APSP is one of the most basic and extensively studied dynamic graph problems. A long line of work culminated in the algorithm of [Chechik, FOCS 2018] with near optimal guarantees: for any constant $0 < \epsilon \leq 1$ and parameter $k \geq 1$, the algorithm achieves approximation factor $(2+\epsilon)k-1$, and total update time $O(mn^{1/k+o(1)}\log(nL))$, where L is the ratio of longest to shortest edge lengths. Unfortunately, as much of prior work, the algorithm is randomized and needs to assume an *oblivious adversary*; that is, the input edge-deletion sequence is fixed in advance and may not depend on the algorithm's behavior.

In many real-world scenarios, and in applications of APSP to static graph problems, it is crucial that the algorithm works against an *adaptive adversary*, where the edge deletion sequence may depend on the algorithm's past behavior arbitrarily; ideally, such an algorithm should be *deterministic*. Unfortunately, unlike the oblivious-adversary setting, its adaptive-adversary counterpart is still poorly understood. For unweighted graphs, the algorithm of [Henzinger, Krinninger and Nanongkai, FOCS '13, SICOMP '16] achieves a $(1+\epsilon)$-approximation with total update time $\tilde{O}(mn/\epsilon)$; the best current total update time guarantee of $n^{2.5+O(\epsilon)}$ is achieved by the recent deterministic algorithm of [Chuzhoy, Saranurak, SODA'21], with $2^{O(1/\epsilon)}$-multiplicative and $2^{O(\log^{3/4} n/\epsilon)}$-additive approximation. To the best of our knowledge, for arbitrary non-negative edge weights, the fastest current adaptive-update algorithm has total update time $O(n^3 \log L/\epsilon)$, achieving a $(1+\epsilon)$-approximation. Even if we are willing to settle for any $o(n)$-approximation factor, no currently known algorithm has a better than $\Theta(n^3)$ total update time in weighted graphs and better than $\Theta(n^{2.5})$ total update time in unweighted graphs. Several conditional lower bounds suggest that no algorithm with a sufficiently small approximation factor can achieve an $o(n^3)$ total update time.

Our main result is a deterministic algorithm for decremental APSP in undirected edge-weighted graphs, that, for any $\Omega(1/\log\log m) \leq \epsilon < 1$, achieves approximation factor $(\log m)^{2^{O(1/\epsilon)}}$, with total update time $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)} \cdot \log L\right)$. In particular, we obtain a $(\operatorname{poly}\log m)$-approximation in time $\widetilde{O}(m^{1+\epsilon})$ for any constant ϵ, and, for any slowly growing function $f(m)$, we obtain $(\log m)^{f(m)}$-approximation in time $m^{1+o(1)}$. We also provide an algorithm with similar guarantees for decremental Sparse Neighborhood Covers.

[*]A preliminary version appeared in STOC 2021

[†]Toyota Technological Institute at Chicago. Email: `cjulia@ttic.edu`. Supported in part by NSF grants CCF-1616584 and CCF-2006464.

Contents

1 Introduction

We study the decremental All-Pairs Shortest-Paths (APSP) problem in weighted undirected graphs. In this problem, we are given as input an undirected graph G with lengths $\ell(e) \geq 1$ on its edges, that undergoes an online sequence of edge deletions. The goal is to support (approximate) shortest-path queries shortest-path-query(x, y): given a pair x, y of vertices of G, return a path connecting x to y, whose length is within factor α of the length of the shortest x-y path in G, where α is the *approximation factor* of the algorithm. We also consider approximate distance queries, dist-query(x, y): given a pair x, y of vertices of G, return an estimate $\mathsf{dist}'(x, y)$ on the distance $\mathsf{dist}_G(x, y)$ between x and y in graph G, such that $\mathsf{dist}_G(x, y) \leq \mathsf{dist}'(x, y) \leq \alpha \cdot \mathsf{dist}_G(x, y)$. APSP is one of the most basic and extensively studied problems in dynamic algorithms, and in graph algorithms in general. Algorithms for this problem often serve as building blocks in designing algorithms for other graph problems, in both the classical static and the dynamic settings. Throughout, we denote by m and n the number of edges and the number of vertices in the initial graph G, respectively, and by L the ratio of largest to smallest edge length. In addition to the approximation factor of the algorithm, two other central measures of its performance are: *query time* – the time it takes to process a single query; and *total update time* – the total time that the algorithm takes, over the course of the entire update sequence, to maintain its data structures. Ideally, we would like the total update time of the algorithm to be close to linear in m, and the query time for shortest-path-query to be bounded by $\widetilde{O}(|E(P)|)$, where P is the path that the algorithm returns.

A straightforward algorithm for the decremental APSP problem is the following: every time a query shortest-path-query(x, y) arrives, compute the shortest x-y path in G from scratch. This algorithm solves the problem exactly, but it has query time $\Theta(m)$. Another approach is to rely on *spanners*. A spanner of a dynamic graph G is another dynamic graph $H \subseteq G$, with $V(H) = V(G)$, such that the distances between the vertices of G are approximately preserved in H; ideally a spanner H should be very sparse. For example, a work of [BKS12] provides a randomized algorithm that maintains a spanner of a fully dynamic n-vertex graph G (that may undergo both edge deletions and edge insertions), that, for any parameter k, achieves approximation factor $(2k - 1)$, has expected amortized update time $O(k^2 \log^2 n)$ per update operation, and expected spanner size $O(kn^{1+1/k} \log n)$. A recent work of [BBG$^+$20] provides a randomized algorithm for maintaining a spanner of a fully dynamic n-vertex graph G with approximation factor $O(\operatorname{poly}\log n)$ and total update time $\widetilde{O}(m^*)$, where m^* is the total number of edges ever present in G; the number of edges in the spanner H is always bounded by $O(n \operatorname{poly}\log n)$. One significant advantage of this algorithm over the algorithm of [BKS12] is that, unlike the algorithm of [BKS12], it can withstand an adaptive adversary; we provide additional discussion of oblivious versus adaptive adversary below. An algorithm for the APSP problem can naturally build on such constructions of spanners: given a query shortest-path-query(x, y) or dist-query(x, y), we simply compute the shortest x-y path in the spanner H. For example, the algorithm for graph spanners of [BBG$^+$20] implies a randomized $\operatorname{poly}\log n$-approximation algorithm for APSP that has $O(m \operatorname{poly}\log n)$ total update time. A recent work of [BHG$^+$20] provides additional spanner-based algorithms for APSP. Unfortunately, it seems inevitable that this straightforward spanner-based approach to APSP must have query time $\Omega(n)$ for both shortest-path-query and dist-query.

In this paper, our focus is on developing algorithms for the APSP problem, whose query time is $\widetilde{O}(|E(P)|)$ for shortest-path-query, where P is the path that the query returns, and $O(\operatorname{poly}\log(mL))$ for dist-query. There are several reasons to strive for these faster query times. First, we typically want responses to the queries to be computed as fast as possible, and the above query times are close to the fastest possible. Second, obtaining $\widetilde{O}(|E(P)|)$ query time for shortest-path-query is often crucial to obtaining fast algorithms for classical (static) graph problems that use algorithms for APSP as a subroutine. We provide an example of such an application to (static) Maximum Multicommodity Flow/

Minimum Multicut in uncapacitated graphs in Section 10.

We distinguish between dynamic algorithms that work against an *oblivious adversary*, where the input sequence of edge deletions is fixed in advance and may not depend on the algorithm's past behavior, and algorithms that work against an *adaptive adversary*, where the input update sequence may depend on the algorithm's past responses and inner states arbitrarily. We refer to the former as *oblivious-update* and to the latter as *adaptive-update* algorithms. We note that any deterministic algorithm for the APSP problem is an adaptive-update algorithm by definition.

The classical data structure of Even and Shiloach [ES81, Din06, HK95], that we refer to as ES-Tree throughout the paper, implies an exact deterministic algorithm for decremental unweighted APSP with $O(mn^2)$ total update time, and the desired $O(|E(P)|)$ query time for shortest-path-query, where P is the returned path. Short of obtaining an exact algorithm for APSP, the best possible approximation factor one may hope for is $(1+\epsilon)$, for any ϵ. A long line of work [BHS07, RZ12, HKN16, Ber16] is dedicated to this direction. The fastest algorithms in this line of work, due to Henzinger, Krinninger, and Nanongkai [HKN16], and due to Bernstein [Ber16] achieve total update time $\tilde{O}(mn/\epsilon)$; the former algorithm is deterministic but only works in unweighted undirected graphs, while the latter algorithm works in directed weighted graphs, with an overhead of $\log L$ in the total update time, but can only handle an oblivious adversary. Unfortunately, known conditional lower bounds show that these algorithms are likely close to the best possible. Specifically, Dor, Halperin and Zwick [DHZ00], and Roddity and Zwick [RZ11] showed that, assuming the Boolean Matrix Multiplication (BMM) conjecture[1], for any $\alpha, \beta \geq 1$ with $2\alpha+\beta < 4$, no combinatorial algorithm for APSP achieves a multiplicative α and additive β approximation, with total update time $O(n^{3-\delta})$ and query time $O(n^{1-\delta})$, for any constant $0 < \delta < 1$. Henzinger et al. [HKNS15] generalized this result to show the same lower bounds for all algorithms and not just combinatorial ones, assuming the Online Boolean Matrix-Vector Multiplication (OMV) conjecture[2]. The work of Vassilevska Williams and Williams [WW18], combined with the work of Roddity and Zwick [RZ11], implies that obtaining such an algorithm would lead to subcubic-time algorithms for a number of important static problems on graphs and matrices.

Due to these negative results, much work on the APSP problem inevitably focused on higher approximation factors. In this regime, the oblivious-update setting is now reasonably well understood. A long line of work [BR11, HKN16, ACT14, FHN14a] recently culminated with a randomized algorithm of Chechik [Che18], that, for any integer $k \geq 1$ and parameter $0 < \epsilon < 1$, obtains a $((2+\epsilon)k-1)$-approximation, with total update time $O(mn^{1/k+o(1)} \log L)$, when the input graph is weighted and undirected. This result is near-optimal, as all its parameters almost match the best static algorithm of [TZ01]. We note that this result was recently slightly improved by [ŁN20], who obtain total update time $O(mn^{1/k} \log L)$, and improve query time for dist-query.

In contrast, progress in the adaptive-update setting has been much slower. Until recently, the fastest adaptive-update algorithm for **unweighted** graphs, due to Henzinger, Krinninger, and Nanongkai [HKN16], only achieved an $\tilde{O}(mn/\epsilon)$ total update time (for approximation factor $(1+\epsilon)$); the algorithm was recently significantly simplified by Gutenberg and Wulff-Nilsen [GWN20]. A recent work of [CS21] provided a deterministic algorithm for unweighted undirected graphs, that, for any parameter $1 \leq k \leq o(\log^{1/8} n)$, in response to query shortest-path-query(x, y), returns a path of length at most $3 \cdot 2^k \cdot \mathsf{dist}_G(x, y) + 2^{(O(k \log^{3/4} n)}$, with query time $O(|E(P)| \cdot n^{o(1)})$ for shortest-path-query, and total update time $n^{2.5+2/k+o(1)}$. To the best of our knowledge, the fastest current adaptive-update algorithm for **weighted** graphs has total update time $O(n^3 \log L/\epsilon)$ and approximation factor $(1+\epsilon)$ (see [KŁ19]).

[1]The conjecture states that there is no "combinatorial" algorithm for multiplying two Boolean matrices of size $n \times n$ in time $n^{3-\delta}$ for any constant $\delta > 0$.

[2]The conjecture assumes that there is no $n^{3-\delta}$-time algorithm, for any constant $0 < \delta < 1$, for the OMV problem, in which the input is a Bollean $(n \times n)$ matrix, with n Boolean dimension-n vectors $v_1, \ldots, v_n$ arriving online. The algorithm needs to output Mv_i immediately after v_i arrives.

Interestingly, even if we allow an $o(n)$-approximation factor, no adaptive-update algorithms with better than $\Theta(n^3)$ total update time and better than $\Theta(n)$ query time for shortest-path-query and dist-query are currently known for weighted undirected graphs, and no adaptive-update algorithms with better than $\Theta(n^{2.5})$ total update time and better than $\Theta(n)$ query time are currently known for unweighted undirected graphs. Moreover, even for the seemingly simpler Single-Source Shortest Path problem (SSSP), where all queries must be between a pre-specified source vertex s and another arbitrary vertex of G, no algorithms achieving a better than $\Theta(n^2)$ total update time, and better than $\Theta(n)$ query time for shortest-path-query are known. To summarize, ideally we would like an algorithm for decremental APSP in weighted undirected graphs that achieves the following properties:

- it can withstand an adaptive adversary (and is ideally deterministic);
- it has query time $\tilde{O}(|E(P)|)$ for shortest-path-query, where P is the returned path, and query time $\tilde{O}(1)$ for dist-query;
- it has near-linear in m total update time; and
- it has a reasonably low approximation factor (ideally, polylogarithmic or constant).

Our main result comes close to achieving all these properties. Specifically, we provide a *deterministic* algorithm for APSP in weighted undirected graphs. For any precision parameter $\Omega(1/\log\log m) < \epsilon < 1$, the algorithm achieves approximation factor $(\log m)^{2^{O(1/\epsilon)}}$, with total update time:

$$O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)} \cdot \log L\right).$$

The query time for processing dist-query is $O(\log m \log\log L)$, and the query time for shortest-path-query is $O(|E(P)|) + O(\log m \log\log L)$, where P is the returned path. In particular, by letting ϵ be a small enough constant, we obtain a $O(\operatorname{poly}\log m)$-approximation with total update time time $O(m^{1+\delta})$, for any constant $0 < \delta < 1$, and by letting $1/\epsilon$ be a slowly growing function of m (for example, $1/\epsilon = O(\log(\log^* m)))$, we obtain an approximation factor $(\log m)^{O(\log^* m)}$, and total update time $O(m^{1+o(1)})$.

In fact we design an algorithm for a more general problem: *dynamic Sparse Neighborhood Cover*. Given a graph G with lengths on edges, a vertex $v \in V(G)$, and a distance parameter D, we denote by $B_G(v, D)$ the *ball of radius D around v*, that is, the set of all vertices u with $\mathsf{dist}_G(v, u) \le D$. Suppose we are given a static graph G with non-negative edge lengths, a distance parameter D (that we call *target distance threshold*), and a desired approximation factor α. A $(D, \alpha \cdot D)$-*neighborhood cover* for G is a collection $\mathcal{C}$ of vertex-induced subgraphs of G (that we call *clusters*), such that, for every vertex $v \in V(G)$, there is some cluster $C \in \mathcal{C}$ with $B_G(v, D) \subseteq V(C)$. Additionally, we require that for every cluster $C \in \mathcal{C}$, for every pair $x, y \in V(C)$ of its vertices, $\mathsf{dist}_G(x, y) \le \alpha \cdot D$; if this property holds, then we say that $\mathcal{C}$ is a *weak* $(D, \alpha \cdot D)$-neighborhood cover of G. If, additionally, the diameter of every cluster $C \in \mathcal{C}$ is bounded by $\alpha \cdot D$, then we say that $\mathcal{C}$ is a *strong* $(D, \alpha \cdot D)$-neighborhood cover of G. Ideally, it is also desirable that the neighborhood cover is *sparse*, that is, every edge (or every vertex) of G only lies in a small number of clusters of $\mathcal{C}$. For this static setting of the problem, the work of [AP90, ABCP98] provides a deterministic algorithm that produces a strong $(D, O(D \log n))$-neighborhood cover of graph G, where every edge lies in at most $O(\log n)$ clusters, with running time $\widetilde{O}(|E(G)| + |V(G)|)$.

In this paper we consider a *decremental* version of the problem, in which the input graph G undergoes an online sequence of edge deletions. We are required to maintain a weak $(D, \alpha \cdot D)$-neighborhood cover $\mathcal{C}$ of the graph G, and we require that the clusters in $\mathcal{C}$ may only be updated in a specific fashion: once an initial neighborhood cover $\mathcal{C}$ of G is computed, we are only allowed to delete edges or vertices from clusters that lie in $\mathcal{C}$, or to add a new cluster C to $\mathcal{C}$, which must be a subgraph of an existing cluster of $\mathcal{C}$. Additionally, we require that the algorithm supports queries $\mathsf{short\text{-}path\text{-}query}(C, v, v')$: given two vertices $v, v' \in V$, and a cluster $C \in \mathcal{C}$ with $v, v' \in C$, return a path P in the current graph G, of length at most $\alpha \cdot D$ connecting v to v' in G, in time $O(|E(P)|)$. The algorithm must also maintain, for every vertex $v \in V(G)$, a cluster $C = \mathsf{CoveringCluster}(v)$ in $\mathcal{C}$, with $B_G(v, D) \subseteq V(C)$. Lastly, we require that the neighborhood cover is *sparse*, namely, for every vertex v of G, the total number of clusters of $\mathcal{C}$ to which v may ever belong over the course of the algorithm is small. It is not hard to verify that an algorithm for the decremental Sparse Neighborhood Cover problem that we just defined immediately implies an algorithm for decremental APSP with the same approximation factor, and the same total update time (to within $O(\log L)$-factor). We provide a deterministic algorithm for the dynamic Sparse Neighborhood Cover problem with approximation factor $\alpha = O\left((\log m)^{2^{O(1/\epsilon)}}\right)$, and total update time $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)}\right)$. Our algorithm ensures that, for every vertex $v \in V(G)$, the total number of clusters of $\mathcal{C}$ that v ever belongs to is bounded by $m^{O(1/\log\log m)}$. We note that algorithms for static Sparse Neighborhood Covers have found many applications in the area of graph algorithms, and so we believe that our algorithm for dynamic Sparse Neighborhood Cover is interesting in its own right. A Sparse Neighborhood Cover for a dynamic graph G naturally provides an emulator for G. If graph G is decremental, then, while the edges may sometimes be inserted into the emulator (when a new cluster is added to the neighborhood cover $\mathcal{C}$), due to the restrictions that we impose on the types of allowed updates to the clusters of $\mathcal{C}$, such edge insertions are limited to very specific types, and so they are relatively easy to deal with. This allows us to compose emulators given by the neighborhood covers recursively. We note that the idea of using clustering of a dynamic graph G in order to construct an emulator was used before (see e.g. the constructions of [FG19, CZ20, FGH20] of dynamic low-stretch spanning trees). In several of these works, a family of clusters of a dynamic graph G is constructed and maintained, and the restrictions on the allowed updates to the cluster family are similar to the ones that we impose; it is also observed in several of these works that with such restrictions one can naturally compose the resulting emulators recursively – an approach that we follow here as well. However, neither of these algorithms provide neighborhood covers, and in fact the clusters that are maintained for each distance scale are disjoint (something that cannot be achieved by neighborhood covers). Additionally, all the above-mentioned algorithms are randomized and assume an oblivious adversary. On the other hand, the algorithms of [HKN16, GWN20] implicitly provide a deterministic algorithm for maintaining a neighborhood cover of a dynamic graph. However, these algorithms have a number of drawbacks: first, the running time for maintaining the neighborhood cover is too prohibitive (the total update time is $O(mn)$). Second, the neighborhood cover maintained is not necessarily sparse; in fact a vertex may lie in a very large number of resulting clusters. Lastly, clusters that join the neighborhood cover as the algorithm progresses may be arbitrary. The restriction that, for every cluster C added to the neighborhood cover $\mathcal{C}$, there must be a cluster C' containing C that already belongs to $\mathcal{C}$, seems crucial in order to allow an easy recursive composition of emulators obtained from the neighborhood covers, and the requirement that the neighborhood cover is sparse is essential for bounding the sizes of the graphs that arise as the result of such recursive compositions.

We provide an application of our algorithm for the APSP problem: a deterministic algorithm for Maximum Multicommodity Flow and Minimum Multicut in unit-capacity graphs. In both problems, the input is an undirected n-vertex m-edge graph G, and a collection $\mathcal{M} = \{(s_1, t_1), \ldots, (s_k, t_k)\}$ of pairs of its vertices, called *demand pairs*. In the Maximum Multicommodity Flow problem, the goal is to send maximum amount of flow between the demand pairs, such that the total amount of flow traversing each edge is at most 1. We denote by $\mathsf{OPT}_{\mathsf{MCF}}$ the value of the optimal solution to this problem. In the Minimum Multicut problem, given a graph G and a collection $\mathcal{M}$ of demand pairs as before, the goal is to select a minimum-cardinality subset $E' \subseteq E(G)$ of edges, such that, for all $1 \leq i \leq k$, vertices s_i and

t_i lie in different connected components of $G \setminus E'$. We use the standard primal-dual technique-based algorithm of [GK98, Fle00], that can equivalently be viewed as an application of the multiplicative weight update paradigm [AHK12], which essentially reduces the Multicommodity Flow problem to decremental APSP; this reduction was first discovered by [Mad10]. Plugging in our algorithm for APSP, we obtain a deterministic algorithm for Maximum Multicommodity Flow, that, for any $0 < \epsilon < 1$, achieves approximation factor $O\left((\log m)^{2^{O(1/\epsilon)}}\right)$, and has running time $\tilde{O}\left(m^{1+O(\epsilon)}(\log m)^{2^{O(1/\epsilon)}} + k/\epsilon\right)$. The algorithm also provides an integral solution to the Maximum Multicommodity Flow problem with congestion $O(\log n)$, and a fractional solution to the standard LP-relaxation for Minimum Multicut. Using the standard ball-growing technique of [LR99, GVY95], we then obtain an algorithm for Minimum Multicut, with the same asymptotic running time, and similar approximation factor. The fastest previous approximation algorithms for Maximum Multicommodity Flow, achieving $(1+\epsilon)$-approximation, have running times $O(k^{O(1)} \cdot m^{4/3}/\epsilon^{O(1)})$ [KMP12] and $\widetilde{O}(mn/\epsilon^2)$ [Mad10]; we are not aware of any algorithms that achieve a faster running time with possibly worse approximation factors, and we are not aware of any fast algorithms for the Minimum Multicut problem. The best polynomial-time algorithm for Minimum Multicut, due to [LR99, GVY95], achieves an $O(\log n)$-approximation.

Before we discuss our results and techniques in more detail, we provide some additional background on related work.

1.1 Other Related Work

APSP on Expanders. A very interesting special case of the APSP problem is APSP on expanders. In this problem, we are given an initial graph G that is a φ-expander. Graph G undergoes a sequence of edge deletions and isolated vertex deletions, that arrive in batches. We are guaranteed that, after each such batch of updates, the resulting graph G remains an $\Omega(\varphi)$-expander. As in the general APSP problem, the goal is to support approximate shortest-path-query in graph G. This problem is especially interesting for several reasons. First, it seems to be a relatively simple special case of the APSP problem, and, if our goal is to obtain better algorithms for general APSP, solving the problem in expander graphs is a natural starting step. Second, this problem arises in various algorithms for *static* cut and flow problems, and seems to be intimately connected to efficient implementations of the Cut-Matching game of [KRV09], which is a central tool in the design of fast algorithms for cut and flow problems (see, e.g. [CGL+19]). Third, expander graphs are increasingly becoming a central tool for designing algorithms for various dynamic graph problems, and obtaining good algorithms for APSP on expanders will likely become a powerful tool in the toolkit of algorithmic techniques in this area. A recent work of [CS21], building on [CGL+19], implies a deterministic algorithm for APSP in expanders with approximation factor $O\left(\Delta^2(\log n)^{O(1/\epsilon^2)}/\varphi\right)$, query time $O(|E(P)|)$ for shortest-path-query, where P is the returned path, and total update time $O\left(n^{1+O(\epsilon)}\Delta^7(\log n)^{O(1/\epsilon^2)}/\varphi^5\right)$; here, Δ is the maximum vertex degree of G, φ is its expansion, and ϵ is a given precision parameter[3]. In fact, algorithms in this paper also use this algorithm for APSP in expanders as a subroutine.

Single-Source Shortest Paths. Single-Source Shortest Paths (SSSP) is a special case of APSP, where all queries must be between a fixed source vertex s and arbitrary other vertices in the graph G. This problem has also been studied extensively. Algorithms for decremental SSSP are a well-established tool in the design of fast algorithms for various variants of maximum s-t flow and minimum s-t cut problems (see, e.g. [Mad10, CK19, CS21]).

[3]The work of [CS21] only explicitly provides such an algorithm for a specific setting of the parameter ϵ, but it is easy to see that the same algorithm works for the whole range of values of ϵ. We prove this in Theorem 5.3 for completeness.

In the oblivious-adversary setting, our understanding of the problem is almost complete: a sequence of works [FHN14b, BR11, FHN14a] has led to a $(1+\epsilon)$-approximation algorithm, that achieves total update time $O(m^{1+o(1)}\log L)$, which is close to the best possible. The query time of the algorithm is also near optimal: query time for dist-query is $\operatorname{poly}\log n$, and query time for shortest-path-query is $\tilde{O}(|E(P)|)$, where P is the returned path. Conditional lower bounds of [DHZ00, RZ11] (that are based on the Boolean Matrix Multiplication conjecture) and of [HKNS15] (based on the Online Matrix-vector Multiplication conjecture), show that no algorithm that solves the problem exactly can simultaneously achieve an $O(n^{1-\delta})$ query time, and $O(n^{3-\delta})$ total update time, for any constant $\delta > 0$, in graphs with $m = \Theta(n^2)$. The work of Vassilevska Williams and Williams [WW18], combined with the work of Roddity and Zwick [RZ11], implies that obtaining an exact algorithm with similar total update time and query time would lead to subcubic-time algorithms for a number of important static problems on graphs and matrices. This shows that the above oblivious-update algorithm is likely close to the best possible.

For the adaptive-update setting, the progress has been slower. It is well known that the classical ES-Tree data structure of Even and Shiloach [ES81, Din06, HK95], 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. Recently, Bernstein and Chechik [BC16, Ber17, BC17], provided algorithms with total update time $\tilde{O}(n^2\log L)$ and $\tilde{O}(n^{5/4}\sqrt{m}) \leq \tilde{O}(mn^{3/4})$, while Gutenberg and Wulff-Nielsen [GWN20] showed an algorithm with $O(m^{1+o(1)}\sqrt{n})$ total update time. Unfortunately, all these algorithms only support distance queries, and they cannot handle shortest-path queries. This problem was recently addressed by [CK19, CS21], leading to a deterministic algorithm with total update time $O(n^{2+o(1)}\log L/\epsilon^2)$, that achieves a $(1+\epsilon)$-approximation factor, and has query time $O(|E(P)|\cdot n^{o(1)}\log\log L)$ for shortest-path-query. Lastly, the work of [BBG$^+$20] on dynamic spanners also provides a randomized adaptive-update $(1+\epsilon)$-approximation algorithm with total update time $O(m\sqrt{n})$, and query time $\tilde{O}(n)$. As mentioned already, they also provide an algorithm for dynamic spanners, leading to a $\operatorname{poly}\log n$-approximation algorithm with total update time $O(m\operatorname{poly}\log n)$ for APSP, and hence for SSSP, with query time $\tilde{O}(n)$. To the best of our knowledge, our result for the APSP problem is also the first adaptive-adversary algorithm for SSSP with near-linear total update time, that achieves an approximation that is below $\Theta(n)$, and query time $\tilde{O}(|E(P)|)$ for shortest-path-query. We now discuss our results and techniques in more detail.

1.2 Our Results and Techniques

Our main result is a deterministic algorithm for decremental APSP, that is summarized in the following theorem.

Theorem 1.1 *There is a deterministic algorithm, that, given an m-edge graph G with length $\ell(e) \geq 1$ on its edges, that undergoes an online sequence of edge deletions, together with a parameter $c/\log\log m < \epsilon < 1$ for some large enough constant c, supports approximate* shortest-path-query *queries and* dist-query *queries with approximation factor $O\left((\log m)^{2^{O(1/\epsilon)}}\right)$. The query time for processing* dist-query *is $O(\log m\log\log L)$, and the query time for processing* shortest-path-query *is $O(|E(P)|)+O(\log m\log\log L)$, where P is the returned path, and L is the ratio of longest to shortest edge length. The total update time of the algorithm is bounded by:*

$$O\left(m^{1+O(\epsilon)}\cdot(\log m)^{O(1/\epsilon^2)}\cdot\log L\right).$$

Our proof exploits the decremental Sparse Neighborhood Cover problem, for which we provide the following algorithm:

Theorem 1.2 *There is a deterministic algorithm, that, given an m-edge graph G with integral lengths $\ell(e) \geq 1$ on its edges, that undergoes an online sequence of edge deletions, together with parameters $c/\log\log m < \epsilon < 1$ for some large enough constant c, and $D \geq 1$, maintains a weak $(D, \alpha \cdot D)$-neighborhood cover $\mathcal{C}$ of G, for $\alpha = O\left((\log m)^{2^{O(1/\epsilon)}}\right)$, and supports queries* $\mathsf{short\text{-}path\text{-}query}(C, v, v')$*: given a cluster $C \in \mathcal{C}$, and two vertices $v, v' \in V(C)$, return a path P connecting v to v' in G, of length at most $\alpha \cdot D$, in time $O(|E(P)|)$. Additionally, for every vertex $v \in V(G)$, the algorithm maintains a cluster $C = \mathsf{CoveringCluster}(v)$ in $\mathcal{C}$, with $B_G(v, D) \subseteq V(C)$. The algorithm starts with $\mathcal{C} = \{G\}$, and the only allowed changes to the clusters in $\mathcal{C}$ are: (i) delete an edge from a cluster $C \in \mathcal{C}$; (ii) delete an isolated vertex from a cluster $C \in \mathcal{C}$; and (iii) add a new cluster C' to $\mathcal{C}$, where $C' \subseteq C$ for some cluster $C \in \mathcal{C}$. The algorithm has total update time $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)}\right)$ and ensures that, for every vertex $v \in V(G)$, the total number of clusters $C \in \mathcal{C}$ to which v ever belongs over the course of the algorithm is at most $m^{O(1/\log\log m)}$.*

We remark that the above theorem requires that we initially set $\mathcal{C} = \{G\}$. Clearly, this initial cluster set $\mathcal{C}$ may not be a valid neighborhood cover of G. Therefore, before the algorithm processes any updates of graph G, it may update this initial cluster set $\mathcal{C}$, via changes of the types that are allowed by the theorem, until it becomes a valid neighborhood cover. We also note that we allow graphs to have parallel edges, so m may be much larger than $|V(G)|$.

Lastly, we provide an efficient algorithm for the Minimum Multicut and Maximum Multicommodity Flow problems in unit-capacity graphs.

Theorem 1.3 *There is a deterministic algorithm, that, given an n-vertex m-edge graph G, a collection $\mathcal{M} = \{(s_1, t_1), \ldots, (s_k, t_k)\}$ of pairs of its vertices, called* demand pairs*, and a precision parameter $c/\log\log m < \epsilon < 1$ for some large enough constant c, computes, in time $\tilde{O}\left(m^{1+O(\epsilon)}(\log m)^{2^{O(1/\epsilon)}} + k/\epsilon\right)$, a solution to the* Maximum Multicommodity Flow *instance $(G, \mathcal{M})$, of value at least $\Omega\left(\mathsf{OPT}_{\mathsf{MCF}}/(\log m)^{2^{O(1/\epsilon)}}\right)$, and a solution to the* Minimum Multicut *instance $(G, \mathcal{M})$, of cost at most $O\left((\log m)^{2^{O(1/\epsilon)}} \cdot \mathsf{OPT}_{\mathsf{MM}}\right)$, where $\mathsf{OPT}_{\mathsf{MCF}}$ and $\mathsf{OPT}_{\mathsf{MM}}$ are optimal solution values to instance $(G, \mathcal{M})$ of* Maximum Multicommodity Flow *and* Minimum Multicut*, respectively.*

The proof of Theorem 1.3 follows immediately from the proof of Theorem 1.2 via standard techniques; see Section 10 for more details. It is also immediate to obtain the proof of Theorem 1.1 from Theorem 1.2 using the standard approach of considering each distance scale separately; see Section 3.4.2 for a formal proof. We now focus on describing our algorithm for the Neighborhood Cover problem from Theorem 1.2, introducing our new ideas and techniques one by one.

Recursive Dynamic Neighborhood Cover. As mentioned already, one advantage of considering the Neighborhood Cover problem is that its solution naturally provides an emulator for the input graph G, which in turn can be used in order to compose algorithms for Neighborhood Cover recursively. In fact, we initially prove a weaker version of Theorem 1.2, by providing an algorithm (that we denote here for brevity by Alg$'$), that achieves a similar approximation factor, but a slower running time of:

$$O\left(m^{1+O(\epsilon)} \cdot \text{poly}(D) \cdot (\log m)^{O(1/\epsilon^2)}\right)$$

(on the positive side, the algorithm maintains a **strong** neighborhood cover of the graph G). Recall that we call the parameter D the *target distance threshold* for the Neighborhood Cover problem instance. We use the recursive composability of Neighborhood Cover in order to obtain the desired running time, as follows[4]. Using standard rescaling techniques, we can assume that $1 \leq D \leq \Theta(m)$. For all $1 \leq i \leq O(1/\epsilon)$, let $D_i = m^{\epsilon i}$. We obtain an algorithm for the Sparse Neighborhood Cover problem for each target distance threshold D_i recursively. For the base of the recursion, when $i = 1$, we simply run Algorithm Alg$'$, to obtain the desired running time of $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)}\right)$. Assume now that we have obtained an algorithm for target distance threshold D_i, that maintains a neighborhood cover $\mathcal{C}_i$ of graph G. In order to obtain an algorithm for target distance threshold D_{i+1}, we construct a new graph H, by starting with $H = G$, deleting all edges of length greater than D_{i+1}, and rounding the lengths of all remaining edges up to the next integral multiple of D_i. Additionally, for every cluster $C \in \mathcal{C}_i$, we add a vertex $u(C)$ (called a supernode), that connects, with an edge of length D_i, to every vertex $v \in V(C) \cap V(G)$. It is not hard to show that this new graph H approximately preserves all distances between the vertices of G, that are in the range $(D_i, D_{i+1}]$. Since the length of every edge in H is an integral multiple of D_i, scaling all edge lengths down by factor D_i does not change the problem. It is then sufficient to solve the Neighborhood Cover problem in the resulting dynamic graph H, with target distance threshold $D_{i+1}/D_i = m^\epsilon$, which can again be done via Algorithm Alg$'$, with total update time $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)}\right)$. The final algorithm for Theorem 1.2 is then obtained by recursively composing Algorithm Alg$'$ with itself $O(1/\epsilon)$ times.

In order to be able to compose algorithms for the Neighborhood Cover problem using the above approach, we define the problem slightly differently, and we call the resulting variation of the problem Recursive Dynamic Neighborhood Cover, or RecDynNC. We assume that the input is a bipartite graph $H = (V, U, E)$, with non-negative edge lengths. Intuitively, the vertices in set V, that we refer to as *regular vertices*, correspond to vertices of the original graph G, while the vertices in set U, that we call *supernodes*, represent some neighborhood cover $\mathcal{C}$ of the graph G that is possibly maintained recursively: $U = \{u(C) \mid C \in \mathcal{C}\}$. (In order to obtain the initial graph H, we subdivide every edge of G by a new regular vertex; we view the original vertices of G as supernodes; and for every vertex $v \in V(G)$, we add a new regular vertex v' that connects to v with a length-1 edge.) In addition to supporting standard edge-deletion and isolated vertex-deletion updates, we require that the algorithm for the RecDynNC problem supports a new update operation, that we call *supernode splitting*[5]. In this operation, we are given a supernode $u \in V(H)$, and a subset E' of edges that are incident to u in graph H. The update creates a new supernode u' in graph H, and, for every edge $e = (u, v) \in E'$, adds a new edge (u', v) of length $\ell(e)$ to H. The purpose of this update operation is to mimic the addition of a new cluster C to $\mathcal{C}$, where $C \subseteq C'$ for some existing cluster $C' \in \mathcal{C}$. The supernode-splitting operation is applied to supernode $u(C')$, with edge set E' containing all edges $(v, u(C'))$ with $v \in V(C)$, and the operation creates a new supernode $u(C)$. Supernode-splitting operation, however, may insert some new edges into the graph H. This creates several difficulties, especially in bounding total update times in terms of number of edges. We get around this problem as follows. Recall that the supernodes in set U generally correspond to clusters in some dynamic neighborhood cover $\mathcal{C}$, that we maintain recursively. We ensure that this neighborhood cover is sparse, that is, every regular vertex may only belong to a small number of such clusters (typically, at most $m^{1/O(\log\log m)}$). This in turn ensures that, in graph H, for every regular vertex $v \in V(H)$, the total number of edges incident to v that ever belong to H is also bounded by $m^{1/O(\log\log m)}$. We refer to this bound as the *dynamic degree bound*, and denote it by μ. Therefore, if we denote by $N(H)$ the number of regular vertices that belong to the initial graph H, then the total number of edges that ever belong to H is bounded by $N(H) \cdot \mu$. This allows us to use the number of regular vertices of H as a proxy to bounding the number of edges

[4]A similar approach of recursive composition of emulators was used in numerous algorithms for APSP; see, e.g. [Che18].

[5]We note that a similar approach to handling cluster-splitting in an emulator that is based on clustering was used before in numerous works, including, e.g., [BC16, Ber17, CK19, CZ20].

in H.

To summarize, the definition of the RecDynNC problem is almost identical to that of the Sparse Neighborhood Cover problem. The main difference is that the input graph now has a specific structure (that is, it is a bipartite graph), and, in addition to edge-deletions, we also need to support isolated vertex deletions and supernode-splitting updates. Additional minor difference is that we only require that the covering properties of the neighborhood cover hold for the regular vertices of H (and not necessarily the supernodes), and we only bound the number of clusters ever containing a vertex for regular vertices (and not supernodes). These are minor technical details that are immaterial to this high-level overview.

Procedure ProcCut and reduction to the MaintainCluster problem. One of the main building blocks of our algorithm is Procedure ProcCut. Suppose our goal is to design an algorithm for the RecDynNC problem on input graph H, with target distance threshold D, and let $\mathcal{C}$ be the neighborhood cover that we maintain. We denote by N the number of regular vertices in the initial graph H, and, for each subgraph $H' \subseteq H$, we denote by $N(H')$ the number of regular vertices in H'. Given a cluster $C \in \mathcal{C}$, and two vertices $x, y \in C$, such that $\mathsf{dist}_C(x, y) > \Omega(D \operatorname{poly}\log N)$, procedure ProcCut produces two vertex-induced subgraphs $C', C'' \subseteq C$, such that $N(C') \leq N(C'')$, $\mathsf{diam}(C') \leq O(D \operatorname{poly}\log N)$, and each of C', C'' contains exactly one of the two vertices x, y. Moreover, it guarantees that, for every vertex $v \in V(C)$, either $B_C(v, D) \subseteq C'$, or $B_C(v, D) \subseteq C''$ holds. We then add C' to $\mathcal{C}$, and update C by deleting edges and vertices from it, until $C = C''$ holds. This procedure is exploited by our algorithm in two ways: first, we compute an initial strong $(D, D \cdot \operatorname{poly}\log N)$-neighborhood cover $\mathcal{C}$ of the input graph H, before it undergoes any updates, by repeatedly invoking this procedure. Later, as the algorithm progresses, and update operations are applied to H, the diameters of some clusters $C \in \mathcal{C}$ may grow. Whenever we identify such a situation, we use Procedure ProcCut in order to cut the cluster C into smaller subclusters. We note that, if C' and C'' are the outcome of applying Procedure ProcCut to cluster C, then we cannot guarantee that the two clusters are disjoint, so they may share edges and vertices. Therefore, a vertex of H may belong to a number of clusters in $\mathcal{C}$. The main challenge in designing Procedure ProcCut is to ensure that every vertex of H only belongs to a small number of clusters (at most $N^{O(1/\log\log N)}$) over the course of the entire algorithm. The procedure uses a carefully designed modification of the ball-growing technique of [LR99] that allows us to ensure this property. We note that several previous works used the ball-growing technique in order to compute and maintain a clustering of a graph. For example, [CZ20] employ this technique in order to maintain clustering at every distance scale. However, the clusters that they maintain at each distance scale are disjoint, and so they can use the standard ball-growing procedure of [LR99] in order to ensure that relatively few edges have endpoints in different clusters. In contrast, in order to maintain a neighborhood cover, we need to allow clusters at each distance scale to overlap. While one can easily adapt the standard ball-growing procedure of [LR99] to still ensure that the total number of edges in the resulting clusters is sufficiently small, this would only ensure that every vertex belongs to relatively few clusters **on average**. It is the strict requirement that **every** vertex may only ever belong to few clusters in the neighborhood cover that makes the design of Procedure ProcCut challenging. We are not aware of any other work that adapted the ball-growing technique to this type of requirement, except for the algorithm of [AP90, ABCP98], who did so in the static setting. It is unclear though how to adapt their techniques to the dynamic setting.

We also use Procedure ProcCut to reduce the RecDynNC problem to a new problem, that we call MaintainCluster. In this problem, we are given some cluster C that was just added to the neighborhood cover $\mathcal{C}$. The goal is to support queries short-path-query(C, v, v'): given a pair $v, v' \in V(C)$ of vertices of C, return a path P connecting v to v' in C, of length at most $\alpha \cdot D$, in time $O(|E(P)|)$. However, the algorithm may, at any time, raise a flag F_C, to indicate that the diameter of C has become too large. When flag F_C is raised, the algorithm must provide two vertices $x, y \in C$, with $\mathsf{dist}_C(x, y) > \Omega(D \operatorname{poly}\log N)$. The algorithm then obtains a sequence of update operations (that we call a *flag-lowering sequence*), at the end of which either x or y are deleted from C, and flag F_C is lowered. Queries short-path-query may only be asked when the flag F_C is down. Once flag F_C is lowered, the algorithm may raise it again immediately, as long as it supplies a new pair $x', y' \in V(C)$ of vertices with $\mathsf{dist}_C(x', y') > \Omega(D \operatorname{poly}\log N)$. Intuitively, once flag F_C is raised, we will simply run Procedure ProcCut on cluster C, with the vertices x, y supplied by the algorithm, and obtain two new clusters C', C''. Assume that C' contains fewer regular vertices than C''. We then add C' to $\mathcal{C}$, and delete edges and vertices from C until $C = C''$ holds, thus creating a flag-lowering update sequence for it. In order to obtain an algorithm for the RecDynNC problem, it is then enough to obtain an algorithm for the MaintainCluster problem. We focus on this problem in the remainder of this exposition.

Pseudocuts, expanders, and their embeddings. The next central tool that we introduce is balanced pseudocuts. Consider a cluster C, for which we would like to solve the MaintainCluster problem, as C undergoes a sequence of online updates, with target distance threshold D. For a given balance parameter ρ, a standard balanced multicut for C can be defined as a set $E' \subseteq E(C)$ of edges, such that every connected component of $C \setminus E'$ contains at most $N(C)/\rho$ regular vertices. We weaken this notion of balanced multicut, and use instead *balanced pseudocuts*. Let $D' = \Theta(D \operatorname{poly}\log N)$. A (D', ρ)-pseudocut in cluster C is a collection E' of its edges, such that, in graph $C \setminus E'$, for every vertex $v \in V(C)$, the ball $B_{C \setminus E'}(v, D')$ contains at most $N(C)/\rho$ regular vertices. In particular, once all edges of E' are deleted from C, if we compute a strong (D, D')-neighborhood cover $\mathcal{C}'$ of C, then we are guaranteed that for all $C' \in \mathcal{C}'$, $N(C') \leq N(C)/\rho$. We note that standard balanced multicuts also achieve this useful property. An advantage of using pseudocuts is that we can design a near-linear time algorithm that computes a (D', ρ)-pseudocut E' in graph C, for $\rho = N^{\epsilon}$, and additionally it computes an expander X, whose vertex set is $\{v_e \mid e \in E''\}$, where $E'' \subseteq E'$ is a large subset of the edges of E', together with an embedding of X into C, via short embedding paths, that causes a low edge-congestion (see Theorem 5.1 for details). This allows us to build on known expander-based techniques in order to design an efficient algorithm for the MaintainCluster problem. Consider the following algorithm, that consists of a number of phases. In every phase, we start by computing a (D', ρ)-pseudocut E' of C, the corresponding expander X, and its embedding into C. Let $E'' \subseteq E'$ be the set of edges e, whose corresponding vertex v_e lies in the expander X, so $V(X) = \{v_e \mid e \in E''\}$. We then use two data structures. The first data structure is an ES-Tree τ, whose root s is a new vertex, that connects to each endpoint of every edge in E'', and has depth $O(D \operatorname{poly}\log N)$. This data structure allows us to ensure that every vertex of C is close enough to some edge of E'', and to identify when this is no longer the case, so that flag F_C is raised. Additionally, we use known algorithms for APSP on expanders, together with the algorithm of [SW19] for expander pruning, in order to maintain the expander X (under update operations performed on the cluster C), and its embedding into C. This allows us to ensure that all edges in E'' remain sufficiently close to each other. These two data structures are sufficient in order to support the short-path-query(C, v, v') queries. If the initial pseudocut E' was sufficiently large, then these data structures can be maintained over a long enough sequence of update operations to cluster C. Once a large enough number of edges are deleted from C, expander X can no longer be maintained, and we recompute the whole data structure from scratch. Therefore, as long as the pseudocut E' that our algorithm computes is sufficiently large (for example, its cardinality is at least $(N(C))^{1-\epsilon}$), we can support the short-path-query(C, v, v') queries as needed, with a very efficient algorithm.

It now remains to deal with the situation where the size of the pseudocut E' is small. One simple way to handle it is to maintain $2|E'|$ ES-Tree data structures, each of which is rooted at an endpoint of a distinct edge of E', and has depth threshold $\Theta(D \operatorname{poly}\log N)$. As long as the root vertex of an ES-Tree τ

remains in the current cluster C, we say that the tree τ *survives*. As long as at least one of the $\mathsf{ES\text{-}Trees}$ rooted at the endpoints of the edges in E' survives, we can support the $\mathsf{short\text{-}path\text{-}query}(C, v, v')$ queries using any such tree. We can also use such a tree in order to detect when the diameter of the cluster becomes too large, and, when this happens, to identify a pair x, y of vertices of C with $\mathsf{dist}_C(x, y)$ sufficiently large. Once every $\mathsf{ES\text{-}Tree}$ that we maintain is destroyed, we are guaranteed that all edges of E' are deleted from C. We can then iteratively apply Procedure $\mathsf{ProcCut}$ in order to further decompose C into a collection of low-diameter clusters (that is, we compute a collection $\mathcal{C}'$ of subgraphs of C, such that $\mathcal{C}'$ is a (D, D')-neighborhood cover for C). Since E' was a (D', ρ)-pseudocut for the original cluster C, we are then guaranteed that every cluster in $\mathcal{C}'$ is significantly smaller than C, and contains at most $N(C)/\rho$ regular vertices. We can then initialize the algorithm for solving the $\mathsf{MaintainCluster}$ problem on each cluster of $\mathcal{C}'$. This approach already gives non-trivial guarantees (though in order to optimize it, we should choose a different threshold for the cardinality of E': if $|E'| > \sqrt{N(C)}$, we should use the expander-based approach, and otherwise we should maintain the $\mathsf{ES\text{-}Tree}$'s). Our rough estimate is that such an algorithm would result in total update time $O\left(m^{1.5+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)}\right)$, but it is still much higher than our desired update time.

In order to achieve our desired near-linear total update time, we exploit again the recursive composability properties of the $\mathsf{RecDynNC}$ problem. Specifically, consider the situation where the pseudocut E' that we have computed is small, that is, $|E'| < (N(C))^{1-\epsilon}$, and consider the graph $H' = C \setminus E'$. For all $1 \leq i \leq \lceil \log D \rceil$, we solve the $\mathsf{RecDynNC}$ problem in graph H' with target distance threshold $D_i = 2^i$ recursively. Fix some index $1 \leq i \leq \lceil \log D \rceil$, and let $\mathcal{C}_i$ be the initial strong $(D_i, D_i \cdot \operatorname{poly}\log N)$-neighborhood cover that this algorithm computes. The properties of the balanced pseudocut ensure that each cluster $C' \in \mathcal{C}_i$ is significantly smaller that C: namely, $N(C') \leq N(C)/\rho \leq (N(C))^{1-\epsilon}$. Therefore, we can solve the $\mathsf{MaintainCluster}$ problem on each such cluster recursively, and we also do so for every cluster that is later added to $\mathcal{C}_i$. Let $\tilde{\mathcal{C}} = \bigcup_i \mathcal{C}_i$ be the dynamic collection of clusters that we maintain.

We use the set $\tilde{\mathcal{C}}$ of clusters in order to construct a *contracted graph* $\hat{H}$. The vertex set of $\hat{H}$ consists of the set S of regular vertices – all regular vertices that serve as endpoints of the edges of E' (the edges of the pseudo-cut); and the set $U' = \left\{u(C') \mid C' \in \tilde{\mathcal{C}}\right\}$ of supernodes. For every edge $e = (u, v) \in E'$, where $v \in S$ is a regular vertex, we add an edge connecting v to every supernode $u(C')$, such that cluster C' contains either v or u. The length of the edge is D_i, where i is the index for which $C' \in \mathcal{C}_i$ holds. It is not hard to show that the distances between the vertices of S are approximately preserved in graph $\hat{H}$. As cluster C undergoes a sequence of update operations, the neighborhood covers $\mathcal{C}_i$ evolve, which in turn leads to changes in the contracted graph $\hat{H}$. However, we ensure that all changes to the neighborhood covers $\mathcal{C}_i$ are only of the types allowed by Theorem 1.2, namely: (i) delete an edge from a cluster of $\mathcal{C}_i$; (ii) delete an isolated vertex from a cluster of $\mathcal{C}_i$; or (iii) add a new cluster C'' to $\mathcal{C}_i$, where $C'' \subseteq C'$ for some cluster $C' \in \mathcal{C}_i$. We are then guaranteed that all resulting changes to graph $\hat{H}$ can be implemented via allowed update operations: namely edge deletions, isolated vertex deletions, and supernode splitting.

We construct two data structures. First, an $\mathsf{ES\text{-}Tree}$ τ, in the graph obtained from C by adding a new source vertex s^*, that connects to every vertex in S with a length-1 edge. The depth of the tree is $O(D \operatorname{poly}\log N)$. This data structure allows us to ensure that every vertex of C is sufficiently close to some vertex of S, and, when this is no longer the case, to raise the flag F_C, and to supply two vertices $x, y \in V(C)$ that are sufficiently far from each other.

The second data structure is obtained by applying the algorithm for the $\mathsf{MaintainCluster}$ problem recursively to the contracted graph $\hat{H}$. This data structure allows us to ensure that all vertices of S are sufficiently close to each other, and, when this is no longer the case, it supplies a pair of vertices $s, s' \in S$, that are sufficiently far from each other in $\hat{H}$, and hence in C. Since we only use this algorithm in the scenario where $|E'| \leq (N(C))^{1-\epsilon}$, we are guaranteed that $|S| \leq (N(C))^{1-\epsilon}$, so graph $\hat{H}$ is significantly smaller than C.

To summarize, in order to solve the MaintainCluster problem in graph C, we use an expander-based approach, as long as the size of the pseudocut E' that our algorithm computes is above $(N(C))^{1-\epsilon}$. Once this is no longer the case, we recursively solve the problem on clusters that are added to the neighborhood covers $\mathcal{C}_i$ of graph $H = C \setminus E'$, for $1 \leq i \leq \lceil \log D \rceil$. This allows us to maintain the neighborhood covers $\{\mathcal{C}_i\}$, which, in turn, allow us to maintain the contracted graph $\hat{H}$. We then solve the MaintainCluster problem recursively on the contracted graph $\hat{H}$. Once all edges of E' are deleted from C, we start the whole algorithm from scratch. Since we ensure that the diameter of C is bounded by D', from the definition of a balanced pseudocut, we are guaranteed that $N(C)$ has decreased by at least a factor N^{ϵ}.

Directions for future improvements. A major remaining open question is whether we can obtain an algorithm for decremental APSP with a significantly better approximation factor, while preserving the near-linear total update time and the near-optimal query time. While it seems plausible that the new tools presented in this paper may lead to an improved $(\log m)^{\text{poly}(1/\epsilon)}$-approximation algorithm with similar running time guarantees, improving the approximation factor beyond the $(\log m)^{\text{poly}(1/\epsilon)}$ barrier seems quite challenging. A necessary first step toward such an improvement is to obtain better approximation algorithms for the decremental APSP problem on expanders. We believe that this is a very interesting problem in its own right, and it is likely that better algorithms for this problem will lead to better deterministic algorithms for basic cut and flow problems, including Minimum Balanced Cut and Sparsest Cut, via the techniques of [CGL+19]. This, however, is not the only barrier to obtaining an approximation factor below $(\log m)^{\text{poly}(1/\epsilon)}$ for decremental APSP in near-linear time. In order to bring the running time of the algorithm for the RecDynNC problem down from $O\left(m^{1+O(\epsilon)} \cdot \text{poly}(D) \cdot (\log m)^{O(1/\epsilon^2)}\right)$ to the desired running time of $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)}\right)$, we recursively compose instances of RecDynNC with each other. This leads to recursion depth $O(1/\epsilon)$, and unfortunately the approximation factor accumulates with each recursive level. If the running time of our basic algorithm for RecDynNC (see Theorem 3.3) can be improved to depend linearly instead of cubically on D, it seems conceivable that one could use the approach of [BC16, Ber17], together with Layered Core Decomposition of [CS21] in order to avoid this recursion (though it is likely that, in the running time of the resulting algorithm, term $m^{1+O(\epsilon)}$ will be replaced with $n^{2+O(\epsilon)}$). Lastly, our algorithm for the MaintainCluster problem needs to call to itself recursively on the contracted graph $\hat{H}$, which again leads to a recursion of depth $O(1/\epsilon)$, with the approximation factor accumulating at each recursive level. One possible direction for reducing the number of the recursive levels is designing an algorithm for computing a pseudocut E', its corresponding expander X, and an embedding of X into the cluster C with a better balance parameter ρ (see Theorem 5.1). We believe that obtaining an analogue of Theorem 5.1 with stronger parameters and faster running time is a problem of independent interest.

1.3 Organization

Most of this paper focuses on the proof of Theorem 1.2. We start with preliminaries in Section 2, and then define the Recursive Dynamic Neighborhood Cover problem (RecDynNC), and state our main result for it in Section 3. We also show in Section 3 that the proofs of Theorem 1.1 and Theorem 1.2 follow from this result. After that, we gradually introduce our new technical tools. In Section 4 we describe and analyze Procedure ProcCut, and use it in order to reduce the RecDynNC problem to a new problem that we define in the same section, called MaintainCluster. In the same section, we show a slow and simple algorithm for the MaintainCluster problem, that we will use as a recursion base. In Section 5 we define balanced pseudocuts, and provide our main algorithm for them, that allows us to compute a pseudocut in a given cluster C, together with the corresponding expander X, and its embedding into C. We also provide an algorithm for APSP on expanders, that is implicit in [CS21]. Lastly, we show that these new tools already lead to a somewhat faster algorithm for the MaintainCluster problem. In Section 6 we define good clusters; intuitively, these are clusters on which we can solve the MaintainCluster problem using the tools that we have developed so far, and we provide an algorithm that does exactly that. We then define a contracted graph and analyze its properties in Section 7, and we complete the algorithm for the MaintainCluster problem, by combining all tools that we have introduced, in Section 8. This completes the proof of Theorem 1.1 and Theorem 1.2. In Section 9 we summarize all main parameters used in the proof. Lastly, in Section 10 we provide our algorithm for Maximum Multicommodity Flow and Minimum Multicut, proving Theorem 1.3.

2 Preliminaries

All logarithms in this paper are to the base of 2. All graphs in this paper are undirected. Graphs may have parallel edges, except for simple graphs, that cannot have them. Throughout the paper, we use a $\tilde{O}(\cdot)$ notation to hide multiplicative factors that are polynomial in $\log m$ and $\log n$, where m and n are the number of edges and vertices, respectively, in the initial input graph.

We follow standard graph-theoretic notation. Given a graph $G = (V, E)$ and two disjoint subsets A, B of its vertices, we denote by $E_G(A, B)$ the set of all edges with one endpoint in A and another in B, and by $E_G(A)$ the set of all edges with both endpoints in A. We also denote by $\delta_G(A)$ the set of all edges with exactly one endpoint in A. For a vertex $v \in V(G)$, we denote by $\delta_G(v)$ the set of all edges incident to v in G, and by $\deg_G(v)$ the degree of v in G. We may omit the subscript G when clear from context. Given a subset $S \subseteq V$ of vertices of G, we denote by $G[S]$ the subgraph of G induced by S.

Given a graph G and a weight function $w : V(G) \to \mathbb{R}$ on its vertices, for a subset $V' \subseteq V(G)$ of its vertices, we denote by $W(V') = \sum_{v \in V'} w(v')$ the total weight of all vertices in V'. Abusing the notation, for a subgraph $C \subseteq G$, we denote the weight of the subgraph $W(C) = \sum_{v \in V(C)} w(v)$.

Given a graph G with lengths $\ell(e) \geq 0$ on edges $e \in E(G)$, for a pair of vertices $u, v \in V(G)$, we denote by $\mathsf{dist}_G(u, v)$ the *distance* between u and v in G, that is, the length of the shortest path between u and v with respect to the edge lengths $\ell(e)$. For a pair S, T of subsets of vertices of G, we define the distance between S and T to be $\mathsf{dist}_G(S, T) = \min_{s \in S, t \in T} \mathsf{dist}_G(s, t)$. For a vertex $v \in V(G)$, and a vertex subset $S \subseteq V(G)$, we also define the distance between v and S as $\mathsf{dist}_G(v, S) = \min_{u \in S} \mathsf{dist}_G(v, u)$. The *diameter* of the graph G, denoted by $\mathsf{diam}(G)$, is the maximum distance between any pair of vertices in G. For a path P in G, we denote its length by $\ell_G(P) = \sum_{e \in E(P)} \ell(e)$. For a vertex $v \in V(G)$ and a distance parameter $D \geq 0$, we denote by $B_G(v, D) = \{u \in V(G) \mid \mathsf{dist}_G(u, v) \leq D\}$ the *ball of radius D around v*. Similarly, for a vertex subset $S \subseteq V(G)$, we let the ball of radius D around S be $B_G(S, D) = \{u \in V(G) \mid \mathsf{dist}_G(u, S) \leq D\}$. We will sometimes omit the subscript G when clear from context.

Neighborhood Covers. Neighborhood Cover is a central notion that we use throughout the paper. We use both a strong and a weak notion of neighborhood covers, that are defined as follows.

Definition (Neighborhood Cover) *Let G be a graph with lengths $\ell(e) > 0$ on edges $e \in E(G)$, let $S \subseteq V(G)$ be a subset of its vertices, and let $D \leq D'$ be two distance parameters. A* weak (D, D')-neighborhood cover *for vertex set S in G is a collection $\mathcal{C} = \{C_1, \ldots, C_r\}$ of vertex-induced subgraphs* *of G called* clusters*, such that:*

- *for every vertex $v \in S$, there is some index $1 \leq i \leq r$ with $B_G(v, D) \subseteq V(C_i)$; and*
- *for all $1 \leq i \leq r$, for every pair $s, s' \in S \cap V(C_i)$ of vertices, $\mathsf{dist}_G(s, s') \leq D'$.*

A set $\mathcal{C}$ of subgraphs of G is a strong (D, D')-neighborhood cover *for vertex set S if it is a weak (D, D')-neighborhood cover for S, and, additionally, for every cluster $C \in \mathcal{C}$, for every pair $s, s' \in S \cap V(C)$ of vertices, $\mathsf{dist}_C(s, s') \leq D'$.*

If the vertex set S is not specified, then we assume that $S = V(G)$.

Dynamic Graphs. Throughout, we consider a graph G that undergoes an online sequence $\Sigma = (\sigma_1, \sigma_2, \ldots)$ of update operations. For now it may be convenient to think of the update operations being edge deletions, though we will consider additional update operations later. After each update operation (e.g. edge deletion), our algorithm will perform some updates to the data structure. We refer to different "times" during the algorithm's execution. The algorithm starts at time 0. For each $t \geq 0$, we refer to "time t in the algorithm's execution" as the time immediately after all updates to the data structures maintained by the algorithm following the tth update $\sigma_t \in \Sigma$ are completed. When we say that some property holds at every time in the algorithm's execution, we mean that the property holds at all times t of the algorithm's execution, but it may not hold, for example, during the procedure that updates the data structures maintained by the algorithm, following some input update operation $\sigma_t \in \Sigma$. For $t \geq 0$, we denote by G^t the graph G at time t; that is, G^0 is the original graph, and for $t \geq 0$, G^t is the graph obtained from G after the first t update operations $\sigma_1, \ldots, \sigma_t$.

Even-Shiloach Trees [ES81, Din06, HK95]. Suppose we are given a graph $G = (V, E)$ with integral lengths $\ell(e) \geq 1$ on its edges $e \in E$, a source s, and a distance bound $D \geq 1$. Even-Shiloach Tree (ES-Tree) algorithm maintains, for every vertex v with $\mathsf{dist}_G(s, v) \leq D$, the distance $\mathsf{dist}_G(s, v)$, under the deletion of edges from G. Moreover, it maintains a shortest-path tree τ rooted at vertex s, that includes all vertices v with $\mathsf{dist}_G(s, v) \leq D$. We denote the corresponding data structure by ES-Tree(G, s, D). The total running time of the algorithm, including the initialization and all edge deletions, is $O(m \cdot D \log n)$, where m is the initial number of edges in G and $n = |V|$. Throughout this paper, we refer to the corresponding data structure as *basic* ES-Tree. We later define a slight generalization of this data structure that allows us to handle some additional update operations. This more general data structure will be referred to as *generalized* ES-Tree.

Cuts, Sparsity and Expanders. Given a graph G, a *cut* in G is a bipartition (A, B) of the set $V(G)$ of its vertices, with $A, B \neq \emptyset$. The *sparsity* of the cut (A, B) is $\varphi_G(A, B) = \frac{|E_G(A,B)|}{\min\{|A|,|B|\}}$. We denote by $\Phi(G)$ the smallest sparsity of any cut in G, and we refer to $\Phi(G)$ as the *expansion* of G.

Definition (Expander) *We say that a graph G is a φ-expander iff $\Phi(G) \geq \varphi$.*

We will repeatedly use the following standard observation, whose proof is included in Section A.1 of Appendix for completeness.

Observation 2.1 *Let G be an n-vertex φ-expander with maximum vertex degree at most Δ. Then for any pair u, v of vertices of G, there is a path connecting u to v that contains at most $\frac{8\Delta \log n}{\varphi}$ edges.*

Expander Pruning. We use an algorithm for expander pruning by [SW19]. We slightly rephrase it so it is defined in terms of graph expansion, instead of conductance that was used in the original paper. For completeness, we include the proof in Section A.2 of Appendix.

Theorem 2.2 (Adaptation of Theorem 1.3 in [SW19]) *There is a deterministic algorithm, that, given an access to the adjacency list of a graph $G = (V, E)$ that is a φ-expander, for some parameter $0 < \varphi < 1$, such that the maximum vertex degree in G is at most Δ, and a sequence $\Sigma = (e_1, e_2, \ldots, e_k)$ of $k \leq \varphi|E|/(10\Delta)$ online edge deletions from G, maintains a vertex set $S \subseteq V$ with the following properties. Let G^i denote the graph $G \setminus \{e_1, \ldots, e_i\}$; 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 the edges of $e_1, \ldots, e_i$ from graph G. Then, for all $1 \leq i \leq k$:*

- $S_{i-1} \subseteq S_i$;
- $|S_i| \leq 8i\Delta/\varphi$;
- $|E(S_i, V \setminus S_i)| \leq 4i$; *and*
- *graph* $G^i[V \setminus S_i]$ *is a* $\varphi/(6\Delta)$*-expander.*

The total running time of the algorithm is $\widetilde{O}(k\Delta^2/\varphi^2)$.

3 Valid Input Structure, Valid Update Operations, Generalized Even-Shiloach Trees, and the Dynamic Recursive Neighborhood Cover Problem

Throughout this paper, we will work with inputs that have a specific structure. The structure is designed in a way that will allow us to naturally compose different instances recursively, by exploiting the notion of neighborhood covers. In order to avoid repeatedly defining such inputs, we provide a definition here, and then refer to it throughout the paper. In this section we also define the types of update operations that we allow for such inputs, and extend the basic Even-Shiloach Tree data structure to support such updates. Lastly, we formally define the Recursive Dynamic Neighborhood Cover problem (RecDynNC) and state our main result for this problem. We then show that this result immediately implies the proofs of Theorems 1.1 and 1.2.

3.1 Valid Input Structure and Valid Update Operations

We start by defining a valid input structure; the definition is used throughout the paper and is intended as a shorthand for the types of inputs most our subroutines use.

Definition (Valid Input Structure) *A valid input structure consists of a bipartite graph $H = (V, U, E)$, a distance threshold $D > 0$ and integral lengths $1 \leq \ell(e) \leq D$ for edges $e \in E$. The vertices in set V are called* regular vertices *and the vertices in set U are called* supernodes. *We denote a valid input structure by $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E(H)}, D\right)$. If the distance threshold D is not explicitly defined, then we set it to ∞.*

Intuitively, supernodes in set U correspond to clusters in a Neighborhood Cover $\mathcal{C}$ of the vertices in V with some (smaller) distance threshold, that is computed and maintained recursively. Given a valid input structure $\mathcal{I} = \left(H, \{\ell(e)\}_{e \in E(H)}, D\right)$, we will allow the following types of update operations, that we refer to as *valid update operations*:

- **Edge Deletion.** Given an edge $e \in E(H)$, delete e from H.
- **Isolated Vertex Deletion.** Given a vertex $x \in V(H)$ that is an isolated vertex, delete x from H; and
- **Supernode Splitting.** The input of this update operation is a supernode $u \in U$ and a non-empty subset $E' \subseteq \delta_H(u)$ of edges incident to u. The update operation creates a new supernode u', and, for every edge $e = (u, v) \in E'$, it adds a new edge $e' = (u', v)$ of length $\ell(e)$ to the graph H. We will sometimes refer to e' as a *copy of edge* e.

For brevity of notation, we will refer to edge-deletion, isolated vertex deletion, and supernode-splitting operations as *valid update operations.* Notice that the update operations may not create new regular vertices, so vertices may be deleted from the vertex set V, but never added to it. A supernode splitting operation, however, adds a new supernode to the graph H, and also inserts edges into H. Unfortunately, this means that the number of edges in H may grow as the result of the update operations, which makes it challenging to analyze the running times of various algorithms that we run on subgraphs C of H in terms of $|E(C)|$. In order to overcome this difficulty, we use the notion of the *dynamic degree bound.*

Definition (Dynamic Degree Bound) *We say that a valid input structure* $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E(H)}, D\right)$*, undergoing a sequence* Σ *of valid update operations has dynamic degree bound* μ *iff for every regular vertex* $v \in V$*, the total number of edges incident to* v *that are ever present in* H *over the course of the update sequence* Σ *is at most* μ*.*

We will usually denote by $N^0(H)$ the number of regular vertices in the initial graph H. If $(\mathcal{I}, \Sigma)$ have dynamic degree bound μ, then we are guaranteed that the number of edges that are ever present in H over the course of the update sequence Σ is bounded by $N^0(H) \cdot \mu$.

In general, we will always ensure that the dynamic degree bound μ is quite low. It may be convenient to think of it as $m^{o(1)}$, where m is the initial number of edges in the input graph G for the APSP problem. Intuitively, every supernode of graph H represents some cluster C in a $(\hat{D}, \hat{D}')$-neighborhood cover $\mathcal{C}$ of G, for some parameters $\hat{D}, \hat{D}' \ll D$. Typically, each regular vertex of H represents some actual vertex of graph G, and an edge (v, u) is present in H iff vertex v belongs to the cluster C that vertex u represents. Intuitively, we will ensure that the neighborhood cover $\mathcal{C}$ of G is constructed and maintained in such a way that the total number of clusters of $\mathcal{C}$ to which a given regular vertex v ever belongs over the course of the algorithm is very small. This will ensure that the dynamic degree bound for graph H is small as well.

Note that we can assume without loss of generality that every vertex in the original graph H^0 has at least one edge incident to it, as otherwise it is an isolated vertex, and will remain so as long as it lies in H. Moreover, from the definition of a supernode-splitting operation, it may not be applied to an isolated vertex (as we require that the edge set E' is non-empty). Therefore, any isolated vertex of H^0 can be ignored. We will therefore assume from now on that every supernode in the original graph H^0 has degree at least 1. (This assumption is only used for convenience, so that we can bound the total number of vertices in H^0 by $O(|E(H^0)|)$.)

We use the following simple observation to show that the distances in the graph H may not decrease as the result of a valid update operation.

Observation 3.1 *Consider the graph H at any time during the execution of the sequence Σ of valid update operations, and let x, x' be any two vertices of H. Let H' be the graph obtained after a single valid update operation on H. Then $\mathsf{dist}_{H'}(x, x') \geq \mathsf{dist}_H(x, x')$.*

Proof: If the valid update operation is an edge deletion, or an isolated vertex deletion, then the statement is clearly true. Assume now that the valid update operation is supernode-splitting, applied to a supernode u, with the corresponding edge set E'. It is easy to verify that $\mathsf{dist}_{H'}(x, x') = \mathsf{dist}_H(x, x')$, since every path P connecting x to x' in H' can be transformed into a path P' of the same length connecting x to x' in H, by replacing the vertex u' with the vertex u on it (if $u' \in V(P)$; otherwise $P \subseteq H$ holds). $\square$

Throughout the paper, it may sometimes be convenient for us to work with vertex weights. In such cases, we will always let the weight of every supernode be 0, and the weight of every regular vertex be 1. For a subgraph $C \subseteq H$, we can then denote by $W(C)$ the total weight of all vertices in C. From the above discussion, $|E(C)| \leq W(C) \cdot \mu$ always holds. Some of the technical tools that we develop work even when vertex weights are arbitrary. Since we believe that some of these tools are interesting in their own right, and may be useful for future work, we state them in the most general way, with arbitrary non-negative vertex weights. However, when applying these tools to our problem, we will always set vertex weights as described above.

3.2 Generalized Even-Shiloach Trees

In this subsection we show that the basic ES-Tree data structure of [ES81, Din06, HK95] can be extended to the setting of a valid input structure that undergoes valid update operations. We do so in the following theorem, whose proof is standard and is deferred to Section B of Appendix. We note that a similar data structure was used, either explicity or implicitly, in numerous previous papers.

Theorem 3.2 *There is a deterministic algorithm, that we refer to as* generalized ES-Tree*, that, given a valid input structure $\mathcal{I} = \left(H, \{\ell(e)\}_{e \in E(H)}, D\right)$, where graph H undergoes a sequence Σ of valid update operations with dynamic degree bound μ, and given additionally a source vertex $s \in V(H)$, and a distance threshold $D^* > 0$, such that the length of every edge in H is bounded by D^*, supports* SSSP-query *queries: given a vertex $x \in V(H)$, either correctly establish, in time $O(1)$, that $\mathsf{dist}_H(s, x) > D^*$, or return a shortest s-x path P in H, in time $O(|E(P)|)$. The total update time of the algorithm is $\widetilde{O}(N^0 \cdot \mu \cdot D^*)$, where N^0 is the number of regular vertices in the initial graph H.*

3.3 The Recursive Dynamic Neighborhood Cover (RecDynNC) Problem

In this subsection we define the Recursive Dynamic Neighborhood Cover problem, and then state our main technical result that provides a deterministic algorithm for this problem. We then provide an algorithm with faster update time by exploiting the recursive composability properties of this problem. This faster algorithm, in turn, will immediately provide an algorithm for the APSP problem, proving Theorem 1.1, and an algorithm for the Neighborhood Cover problem, proving Theorem 1.2.

Problem Definition. The input to the Recursive Dynamic Neighborhood Cover (RecDynNC) problem is a valid input structure $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E}, D\right)$, where graph H undergoes a sequence Σ of valid update operations with some given dynamic degree bound μ. Additionally, we are given a desired approximation factor α. We assume that we are also given some arbitrary fixed ordering

$\mathcal{O}$ of the vertices of H, and that any new vertex that is inserted into H as the result of supernode-splitting updates appears at the end of the current ordering. The goal is to maintain the following data structures:

- a collection $\mathcal{U}$ of subsets of vertices of graph H, together with a collection $\mathcal{C} = \{H[S] \mid S \in \mathcal{U}\}$ of clusters in H, such that $\mathcal{C}$ is a weak $(D, \alpha \cdot D)$ neighborhood cover for the set V of regular vertices in graph H. For every set $S \in \mathcal{U}$, the vertices of S must be maintained in a list, sorted according to the ordering $\mathcal{O}$;
- for every regular vertex $v \in V$, a cluster $C = \mathsf{CoveringCluster}(v)$, with $B_H(v, D) \subseteq V(C)$;
- for every vertex $x \in V(H)$, a list $\mathsf{ClusterList}(x) \subseteq \mathcal{C}$ of all clusters containing x, and for every edge $e \in E(H)$, a list $\mathsf{ClusterList}(e) \subseteq \mathcal{C}$ of all clusters containing e.

The set $\mathcal{U}$ of vertex subsets must be maintained as follows. Initially, $\mathcal{U} = \left\{V(H^0)\right\}$, where H^0 is the initial input graph H. After that, the only allowed changes to vertex sets in $\mathcal{U}$ are:

- DeleteVertex(S, x): given a vertex set $S \in \mathcal{U}$, and a vertex $x \in S$, delete x from S;
- AddSuperNode(S, u): if u is a supernode that is lying in S that just underwent supernode splitting update, add the newly created supernode u' to S; and
- ClusterSplit(S, S'): given a vertex set $S \in \mathcal{U}$, and a subset $S' \subseteq S$ of its vertices, add S' to $\mathcal{U}$.

We refer to the above operations as *allowed changes to $\mathcal{U}$*. In other words, if we consider the sequence of changes that clusters in $\mathcal{C}$ undergo over the course of the algorithm, the corresponding sequence of changes in vertex sets in $\{U(C) \mid C \in \mathcal{C}\}$ must obey the above rules.

In addition to maintaining the above data structures, an algorithm for the $\mathsf{RecDynNC}$ problem needs to support queries $\mathsf{short\text{-}path\text{-}query}(C, v, v')$: given two **regular** vertices $v, v' \in V$, and a cluster $C \in \mathcal{C}$ with $v, v' \in C$, return a path P in the current graph H, of length at most $\alpha \cdot D$ connecting v to v' in H, in time $O(|E(P)|)$. This completes the definition of the $\mathsf{RecDynNC}$ problem.

Statement of Main Technical Result

Our main technical result is a deterministic algorithm for the $\mathsf{RecDynNC}$ problem, that is summarized in the following theorem.

Theorem 3.3 *There is a deterministic algorithm for the* $\mathsf{RecDynNC}$ *problem, that, on input* $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E}, D\right)$ *undergoing a sequence of valid update operations with dynamic degree bound μ, and a parameter $c/\log\log W < \epsilon < 1$, for some large enough constant c, where W is the number of regular vertices in H at the beginning of the algorithm, achieves approximation factor* $\alpha = (\log(W\mu))^{2^{O(1/\epsilon)}}$*, and has total update time* $O\left(W^{1+O(\epsilon)} \cdot \mu^{2+O(\epsilon)} \cdot D^3 \cdot (\log(W\mu))^{O(1/\epsilon^2)}\right)$*. Moreover, the algorithm ensures that for every regular vertex $v \in V$, the total number of clusters in the neighborhood cover $\mathcal{C}$ that the algorithm maintains, to which vertex v ever belongs over the course of the algorithm is bounded by* $W^{O(1/\log\log W)}$*. It also ensures that the neighborhood cover $\mathcal{C}$ that it maintains is a strong $(D, \alpha \cdot D)$ neighborhood cover for the set V of regular vertices of H.*

The following sections of this paper focus on the proof of Theorem 3.3. Notice however that the running time of the algorithm from Theorem 3.3 is somewhat slow, and in particular it has a high dependence on the distance threshold D, that we would like to avoid. We now show an algorithm that exploits the recursive composability of instances of the $\mathsf{RecDynNC}$ problem in order to do so.

3.4 A Faster Algorithm for RecDynNC and Proofs of Theorems 1.1 and 1.2

We prove the following theorem, that provides a faster algorithm for the RecDynNC problem, and follows from Theorem 3.3.

Theorem 3.4 *There is a deterministic algorithm for the* RecDynNC *problem, that, given a valid input structure* $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E}, D\right)$ *undergoing a sequence of edge-deletion and isolated vertex-deletion operations, with dynamic degree bound 2, and a parameter* $c/\log\log W < \epsilon < 1$*, for some large enough constant* c*, where* W *is the number of regular vertices in* H *at the beginning of the algorithm, achieves approximation factor* $\alpha = (\log W)^{2^{O(1/\epsilon)}}$*, with total update time* $O\left(W^{1+O(\epsilon)} \cdot (\log W)^{O(1/\epsilon^2)}\right)$*. Moreover, the algorithm ensures that for every regular vertex* $v \in V$*, the total number of clusters in the weak neighborhood cover* $\mathcal{C}$ *that the algorithm maintains, to which vertex* v *ever belongs over the course of the algorithm, is bounded by* $W^{O(1/\log\log W)}$*.*

We note that, since the dynamic degree bound is 2, the degree of every regular vertex in the initial graph H is at most 2. Notice that supernode-splitting operations are not allowed in the above theorem, and so the degree of every regular vertex remains at most 2 over the course of the algorithm. Therefore, the number of edges in H is bounded by $2W$ throughout the algorithm.

We start by showing that Theorems 1.1 and 1.2 immediately follow from Theorem 3.4.

3.4.1 Completing the proof of Theorem 1.2

We assume that we are given an m-edge graph G with integral length $\ell(e) \geq 1$ on its edges, that undergoes an online sequence of edge deletions, together with parameters $c'/\log\log m < \epsilon < 1$ for some large enough constant c', and $D \geq 1$. Note that we can assume that G is a connected graph, as otherwise we can run the algorithm on each of its connected components separately, so $|V(G)| \leq m$ holds. We construct a bipartite graph $H = (V, U, E)$ as follows. We start with the graph G, and we let $U = V(G)$ be the set of the supernodes of H. We then subdivide every edge $e \in E(G)$ with a new regular vertex v_e, and we set the lengths of both new edges to be $\ell(e)$. The set V of regular vertices of H is the union of two subsets: a set $\{v_e \mid e \in E(G)\}$ of vertices corresponding to edges of G, and another subset $S = \{x' \mid x \in V(G)\}$ of vertices corresponding to vertices of G. Every vertex $x' \in S$ connects to the corresponding vertex $x \in V(G)$ with a length-1 edge. Once we delete all edges of length greater than $3D$, we obtain a valid input structure $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E}, 3D\right)$. Given an online sequence Σ of edge deletions for graph G, we can produce a corresponding online sequence Σ' of edge deletions and isolated vertex deletions for graph H, as follows: whenever an edge $e \in E(G)$ is deleted from G, we delete its two corresponding edges (that are incident to v_e) from graph H, and we then delete vertex v_e that becomes an isolated vertex. We have therefore obtained an instance $\mathcal{I}$ of the RecDynNC problem, on valid input structure $\mathcal{I}$ that undergoes a sequence of edge-deletion and isolated vertex-deletion operations. Since the degree of every regular vertex in H is at most 2, it is easy to see that H has dynamic degree bound 2. We let $W = |V| \leq 2m$ be the number of regular vertices in H. By letting c' be a large enough constant, we can assume that $c/\log\log W < \epsilon < 1$, where c is the constant from Theorem 3.4. We run the algorithm for the RecDynNC problem from Theorem 3.4 on input $\mathcal{I}$ undergoing the sequence Σ' of update operations. Let $\mathcal{C}$ be the neighborhood cover that the algorithm maintains. We then define a neighborhood cover $\mathcal{C}'$ for graph G as follows. For every cluster $C \in \mathcal{C}$, there is a cluster $C' \in \mathcal{C}'$, which is a subgraph of G induced by vertex set $\{x \in V(G) \mid x' \in V(C)\}$. Recall that cluster set $\mathcal{C}$ is initially defined to be $\mathcal{C} = \{H\}$, so initially, $\mathcal{C}' = \{G\}$ holds. After that, the only changes to vertex sets in $\mathcal{U} = \{U(C) \mid C \in \mathcal{C}\}$ are the allowed changes, that include:

- DeleteVertex(R, x): given a vertex set $R \in \mathcal{U}$, and a vertex $x \in R$, delete x from R; if $x = y'$ for some vertex $y \in V(G)$, then we also delete y from the cluster $C \in \mathcal{C}'$ representing $H[R]$;
- AddSuperNode(R, u): since we do not allow supernode splitting operations, no such updates will be performed; and
- ClusterSplit$(\tilde{R}, R)$: given a vertex set $R \in \mathcal{U}$, and a subset $\tilde{R} \subseteq R$ of its vertices, add $\tilde{R}$ to $\mathcal{U}$. In this case, we create a new cluster in $\mathcal{C}'$ that is a subgraph of G, induced by the set $\{x \in V(G) \mid x' \in R'\}$ of vertices.

Additionally, when an edge e is deleted from G, we need to delete it from every cluster of $\mathcal{C}$ that contains it. The time that is needed to make all these updates to cluster set $\mathcal{C}'$ is subsumed by the time required to maintain cluster set $\mathcal{C}$.

Consider now some vertex $x \in V(G)$. Recall that the algorithm for the $\mathsf{RecDynNC}$ problem maintains a cluster $C \in \mathcal{C}$, with $B_H(x', 3D) \subseteq V(C)$. It is easy to verify that $B_H(x', 3D)$ contains, for every vertex $y \in B_G(x, D)$, the corresponding vertex $y' \in V$. We then set $\mathsf{CoveringCluster}(x) = C'$, where $C' \in \mathcal{C}$ is the cluster corresponding to C.

Lastly, suppose we are given a query $\mathsf{short\text{-}path\text{-}query}(C', x, y)$, where $C' \in \mathcal{C}'$, and $x, y \in V(C)$. We then run query $\mathsf{short\text{-}path\text{-}query}(C, x', y')$ in the data structure maintained by the algorithm for the $\mathsf{RecDynNC}$ problem, obtaining a path P in graph H that connects x' to y' and has length at most $3\alpha D$, for $\alpha = (\log m)^{2^{O(1/\epsilon)}}$. By deleting the first and the last vertex on P, and by suppressing some vertices, we obtain a path P' in graph G, connecting x to y, whose length is at most $D \cdot (\log m)^{2^{O(1/\epsilon)}}$. The running time of this algorithm is $O(|E(P)|)$. The fact that we can support queries $\mathsf{short\text{-}path\text{-}query}(C', x, y)$, together with the above discussion, proves that the cluster set $\mathcal{C}'$ that we maintain is a weak $(D, 3\alpha D)$-neighborhood cover for graph G. Moreover, from Theorem 3.4, the total update time of the algorithm is $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)}\right)$, and we are also guaranteed that for every vertex $v \in V(G)$, the total number of clusters in $\mathcal{C}'$ to which vertex v ever belonged is at most $m^{O(1/\log\log m)}$.

3.4.2 Completing the proof of Theorem 1.1

By using standard scaling and rounding of edge lengths, at the cost of losing a factor 2 in the approximation ratio, we can assume that all edge lengths are integers between 1 and L. We define $\lambda = \lceil \log L \rceil$ distance thresholds $D_1, \ldots, D_\lambda$, where for $1 \leq i \leq \lambda$, $D_i = 2^i$.

For all $1 \leq i \leq \lambda$, we apply the algorithm from Theorem 1.2 to graph G, for distance threshold D_i, the input parameter ϵ, and approximation factor $\alpha = O\left((\log m)^{2^{O(1/\epsilon)}}\right)$. We denote the neighborhood cover that the algorithm maintains by $\mathcal{C}_i$ and the corresponding data structure by $\mathcal{D}_i$. Note that the total update time of this algorithm, over all distance scales D_i is $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)} \cdot \log L\right)$, as required. We now show algorithms for processing $\mathsf{dist\text{-}query}$ and $\mathsf{shortest\text{-}path\text{-}query}$ queries.

Given a query $\mathsf{dist\text{-}query}(x, y)$, for a pair $x, y \in V(G)$ of vertices, we perform a binary search on an integer $1 \leq i \leq \lambda$, such that, if $C = \mathsf{CoveringCluster}(x)$ is a cluster in $\mathcal{C}_i$ that covers x, then $y \in V(C)$, but, if $C' = \mathsf{CoveringCluster}(x)$ is the cluster of $\mathcal{C}_{i-1}$ covering x, then $y \notin C'$. In order to do so, we consider the current guess i' on the integer i, find the cluster $C = \mathsf{CoveringCluster}(x)$ in data structure $\mathcal{D}_{i'}$ in time $O(1)$, and check whether $y \in V(C)$ in time $O(\log m)$. If this is the case, then in our next guess we increase the index i'; otherwise we decrease it. Once the desired index i is found, we return the value $2D_i$. We claim that $D_i/2 \leq \mathsf{dist}_G(x, y) \leq \alpha D_i$. Indeed, from the definition of weak $(D, \alpha D)$-neighborhood cover, since x and y lie in the same cluster of $\mathcal{C}_i$, $\mathsf{dist}_G(x, y) \leq \alpha D_i$. Moreover, since y does not lie in the cluster $C' = \mathsf{CoveringCluster}(x)$ in $\mathcal{C}_{i-1}$, we get that $y \notin B_G(x, D_{i-1})$, and so $\mathsf{dist}_G(x,y) > D_{i-1} = D_i/2$. Since we perform a binary search over $\lambda = \lceil \log L \rceil$ values i, the running time of the algorithm for responding to the query is $O(\log m \log\log L)$.

Consider now a query $\mathsf{shortest\text{-}path\text{-}query}(x,y)$ for a pair $x, y \in V(G)$ of vertices. We start by computing an index i exactly as in $\mathsf{dist\text{-}query}(x,y)$, so that $D_i/2 \leq \mathsf{dist}_G(x,y) \leq \alpha D_i$, together with the cluster $C = \mathsf{CoveringCluster}(x)$ in $\mathcal{C}_i$, so that $y \in C$ holds. This can be done in time $O(\log m \log\log L)$ as before. We then run query $\mathsf{short\text{-}path\text{-}query}(C,x,y)$ in data structure $\mathcal{D}_i$, to compute a path P in graph G, connecting x to y, of length at most $\alpha \cdot D_i \leq 2\alpha \mathsf{dist}_G(x,y) \leq (\log m)^{2^{O(1/\epsilon)}} \cdot \mathsf{dist}_G(x,y)$, in time $O(|E(P)|)$.

Therefore, in order to prove Theorems 1.1 and 1.2, it is enough to prove Theorem 3.4.

3.4.3 Proof of Theorem 3.4

In this subsection we prove Theorem 3.4 using the algorithm from Theorem 3.3. We can assume that graph H has no isolated vertices, as all such vertices can be ignored (e.g. each such vertex can be placed in a separate cluster). For simplicity, we denote by n and m the number of vertices and edges in the initial graph H. Notice that $m \leq 2W$, and $n \leq 3W$ must hold.

We start by showing that, at the cost of losing a factor 2 in the approximation ratio, we can assume that $D = 2n$, and that all edge lengths are integers between 1 and D. We show this using standard arguments. Recall that, since $\mathcal{I}$ is a valid input structure, all edges in H have lengths at most D. Next, we set the length of each edge e to be $\ell'(e) = \lceil n\ell(e)/D \rceil$.

For every pair x, y of vertices, let $\mathsf{dist}'(x,y)$ denote the distance between x and y with respect to the new edge length values. Notice that for every pair x, y of vertices, $\frac{n}{D} \cdot \mathsf{dist}(x,y) \leq \mathsf{dist}'(x,y) \leq \frac{n}{D} \cdot \mathsf{dist}(x,y) + n$, since the shortest x-y path contains at most n edges.

Therefore, if $\mathsf{dist}(x,y) \leq D$, then $\mathsf{dist}'(x,y) \leq 2n$. Moreover, if P is an x-y path with $\ell'(P) \leq \alpha n$, then $\ell(P) \leq \alpha D$ must hold. It is now enough to solve the problem on graph H with the new edge weights $\ell'(e)$ for $e \in E(H)$, and distance bound $D' = 2n$. Therefore, we assume from now on that $D = 2n$, and that all edge lengths are integers between 1 and D.

The main idea of the proof is to apply the algorithm from Theorem 3.3 recursively, for smaller and smaller distance bounds. Specifically, we prove the following lemma by induction.

Lemma 3.5 *There is a universal constant $\tilde{c}$, and a deterministic algorithm for the* $\mathsf{RecDynNC}$ *problem, that, given a valid input structure $\mathcal{I} = \left(H = (V,U,E), \{\ell(e)\}_{e \in E(H)}, D\right)$ undergoing a sequence of edge-deletion and isolated vertex-deletion operations, with dynamic degree bound 2, and a parameter $0 < \epsilon < 1$, such that $D \leq 6W^{\epsilon i}$ holds for some integer i, where W is the number of regular vertices in H at the beginning of the algorithm, achieves approximation factor $\alpha_i = (\log W)^{\tilde{c} i \cdot 2^{\tilde{c}/\epsilon}}$, and has total updtate time at most $\left(\tilde{c}^i \cdot W^{1+\tilde{c}\epsilon} \cdot (\log W)^{\tilde{c}/\epsilon^2}\right)$. Moreover, the algorithm ensures that for every regular vertex $v \in V$, the total number of clusters in the weak neighborhood cover $\mathcal{C}$ that the algorithm maintains, to which vertex v ever belongs over the course of the algorithm, is bounded by $W^{O(1/\log\log W)}$.*

Notice that the above lemma completes the proof of Theorem 3.4, by setting $i = \lceil 1/\epsilon \rceil$. The approximation factor achieved by the algorithm is at most $(\log W)^{2\tilde{c} \cdot 2^{\tilde{c}/\epsilon}/\epsilon} = (\log W)^{2^{O(1/\epsilon)}}$, and its running time is $O\left(W^{1+O(\epsilon)} \cdot (\log W)^{O(1/\epsilon^2)}\right)$. The proof of the lemma is somewhat technical and is deferred to Section C of Appendix. We provide here a short intuitive overview of the proof. The proof is by induction on i, where the base case, with $i = 1$, follows immediately by invoking the algorithm from Theorem 3.3. In order to prove the lemma for some integer $i > 1$, we select a threshold $D' \approx 6W^{\epsilon(i-1)}$; we say that an edge $e \in E$ is long, if its length is greater than D', and it is short otherwise. Let H' be the graph obtained from H after all long edges are deleted from it. We can then use the induction hypothesis in order to maintain a data structure $\mathcal{D}(H')$, solving the RecDynNC problem on graph H', with distance bound $\Theta(D')$, and approximation factor $\alpha_{i-1} = (\log W)^{\tilde{c}(i-1)\cdot 2^{\tilde{c}/\epsilon}}$. Let $\mathcal{C}'$ be the collection of clusters maintained by the algorithm. Next, we define another graph $\hat{H}$, as follows. Initially, we start with $\hat{H} = H$, except that we increase all edge lengths to become integral multiples of D'; this does not distort the lengths of long edges by much, but it may increase lengths of short edges significantly. Additionally, for every cluster $C \in \mathcal{C}'$, we add a supernode $u(C)$ to graph $\hat{H}$, that connects to every regular vertex $v \in V(C)$ with an edge of length D'. Let $\hat{H}^0$ be the initial graph $\hat{H}$, before any updates to graph H. We show that we can use the data structure $\mathcal{D}(H')$, and the corresponding dynamic neighborhood cover $\mathcal{C}'$, in order to produce a sequence of valid update operations for graph $\hat{H}$, so that, if the resulting sequence of update operations is applied to the initial graph $\hat{H}^0$, then the resulting dynamic graph is precisely $\hat{H}$. We also show that distances are approximately preserved in $\hat{H}$: that is, if v, v' are two regular vertices in graph H that lie at distance at most D in H, then the distance between v and v' in $\hat{H}$ is also bounded by $O(D)$. Additionally, we show an algorithm that, given a path P in graph $\hat{H}$, connecting any pair v, v' of its regular vertices, produces a path P' in graph H, connecting the same pair of vertices, such that the length of P' is close to the length of P. Lastly, we scale all edge lengths in graph $\hat{H}$ down by factor D', and use the algorithm from Theorem 3.3 in order to maintain a neighborhood cover $\hat{\mathcal{C}}'$ in the resulting graph $\hat{H}'$, with distance threshold $\hat{D}' = D/D' \leq W^{O(\epsilon)}$. We exploit this data structure (that we denote by $\mathcal{D}(\hat{H}')$), and the fact that $V(H) \subseteq V(\hat{H}')$, in order to maintain neighborhood cover $\mathcal{C}$ for graph H, as follows. Consider the collection of vertex sets $\hat{\mathcal{U}}' = \left\{V(C) \mid C \in \hat{\mathcal{C}}'\right\}$ that is maintained by data structure $\mathcal{D}(\hat{H}')$. For every vertex set $S \in \hat{\mathcal{U}}'$, we define a set $S' = S \cap V(H)$. We then let $\mathcal{C} = \left\{H[S'] \mid S \in \hat{\mathcal{U}}'\right\}$ be the neighborhood cover for graph H. In other words, we maintain the same set of clusters as $\hat{\mathcal{C}}'$, except that we ignore vertices that do not lie in the graph H. This completes the high-level description of the proof of Lemma 3.5. A formal proof appears in Section C of Appendix.

The remainder of this paper focuses on the proof of Theorem 3.3. In subsequent sections, we gradually introduce several new technical tools, which are used in order to define faster and faster algorithms for the RecDynNC problem, building up to the proof of Theorem 3.3.

4 First Set of Tools: Procedure ProcCut, Initial Neighborhood-Cover Decomposition, and Reduction to MaintainCluster Problem

In this section we introduce one of the central tools that we use in order to maintain neighborhood cover – Procedure ProcCut. Intuitively, suppose we are given a valid input structure $\mathcal{I} = \left(H, \{\ell(e)\}_{e\in E(H)}, D\right)$ that undergoes a sequence of valid update operations with dynamic degree bound μ, and our goal is to solve the RecDynNC problem on it, for some approximation factor α. We set vertex weights in the usual way: the weight of every regular vertex is 1, and the weight of each supernode is 0. We assume that we are given some parameter $W \geq W^0(C)$, where $W^0(C)$ is the weight of all vertices in the initial graph C. We will maintain a neighborhood cover $\mathcal{C}$ of H, starting from $\mathcal{C} = \{H\}$. Given some subgraph $C \subseteq H$ (that may lie in the neighborhood cover $\mathcal{C}$), whose diameter is greater than $D' = \Theta(D \log^4 W)$, the procedure produces two new clusters $C', C'' \subseteq C$. It outputs cluster C' explicitly, by listing all its vertices and edges, while cluster C'' is output implicitly, by listing all vertices and edges of $C \setminus C''$. Cluster C' is then added to the neighborhood cover $\mathcal{C}$, and cluster

$C \in \mathcal{C}$ is updated so that $C = C''$ holds. We say that cluster C' was *split off of* C. In order to ensure that $\mathcal{C}$ remains a valid neighborhood cover, we cannot guarantee that the two clusters C' and C'' are disjoint, but we will ensure that they have a small overlap. At the beginning of our algorithm for the RecDynNC problem, we will compute an initial neighborhood cover $\mathcal{C}$ via Algorithm InitNC, that is described later in this section, which iteratively employs Procedure ProcCut on the input graph H. As the algorithm progresses and edges are deleted from H, the diameters of some clusters $C \in \mathcal{C}$ may become large. Intuitively, whenever that happens, we would like to further decompose the cluster C by employing Procedure ProcCut again. In practice, we will define another problem, MaintainCluster, whose goal is to maintain the cluster $C \in \mathcal{C}$ and to support queries short-path-query(C, v, v'): given two **regular** vertices $v, v' \in V(C)$, return a path P in C, of length at most $\alpha \cdot D$ connecting v to v', in time $O(|E(P)|)$. The algorithm for the MaintainCluster problem on cluster C may, at any time, raise a flag F_C to indicate that it can no longer support short-path-query queries because the diameter of C became too large. When flag F_C is raised, the algorithm for the MaintainCluster problem needs to supply two vertices $x, y \in V(C)$ with $\mathsf{dist}_C(x, y) > \Omega(D \log^4 W)$. We will then employ Procedure ProcCut on graph C, possibly multiple times, in order to update the cluster C, so that its diameter falls back below αD. As part of this process, new clusters will be added to the neighborhood cover $\mathcal{C}$. As mentioned already, the clusters in $\mathcal{C}$ may not be disjoint. However, we will ensure that every regular vertex lies in only a small number of such clusters. It is this requirement that makes the description and the analysis of Procedure ProcCut somewhat challenging.

The remainder of this section is organized as follows. We start with some basic definitions and notation in Section 4.1. We describe Procedure ProcCut and provide its analysis in Section 4.2. Then we describe and analyze Procedure InitNC for computing an initial Neighborhood Cover, in Section 4.3. In Section 4.4, we provide a general algorithmic framework for solving the RecDynNC problem that exploits both Procedure InitNC and Procedure ProcCut. The framework assumes that we are given an oracle that flags clusters C whose diameter become too large, and provides witnesses for this, in the form of a pair x, y of vertices of C whose distance in C is large. In Section 4.5 we define a new problem, called MaintainCluster. An algorithm for this problem will be used in order to implement the oracle in the above-mentioned algorithmic framework, and will also be responsible for supporting short-path-query queries within the clusters. We also state our main result for the MaintainCluster problem – namely, a fast dynamic algorithm for it, and complete the proof of Theorem 3.3 using it in Section 4.5. The remainder of the paper then focuses on desiginig a fast algorithm for the MaintainCluster problem. As a first step towards this goal, in Section 4.6 we provide a slow and simple algorithm for the MaintainCluster problem, called AlgSlow, that we will use as a recursion base.

4.1 Basic Definition, Parameters and Notation

Throughout this section, we assume that we are given a valid input structure $\mathcal{I} = \left(H, \{\ell(e)\}_{e \in E(H)}, D\right)$, with $H = (V, U, E)$, that undergoes a sequence $\Sigma = (\sigma_1, \sigma_2, \ldots)$ of valid update operations with dynamic degree bound μ. We define vertex weights in the usual way: the weight of every regular vertex is 1, and the weight of every supernode is 0. We denote by W the total weight of all vertices at the beginning of the algorithm. We assume that we are given a target approximation factor $\alpha \geq \Omega(\log^4 W)$. Our goal is to solve the RecDynNC problem on $\mathcal{I}$ with approximation factor α. Therefore, we will maintain a collection $\mathcal{C}$ of vertex-induced subgraphs of H, that we call clusters. At the beginning of the algorithm, we set $\mathcal{C} = \left\{H^0\right\}$, where H^0 is the initial graph H (but we will may make some additional updates to this initial set of clusters before processing the first update $\sigma_1 \in \Sigma$).

We use the following parameters: $r = 2\left\lceil \frac{\log W}{\log \log W} \right\rceil$, and $\gamma = 2^r = W^{O(1/\log\log W)}$. As mentioned above, we aim to construct and maintain a strong $(D, \alpha D)$-neighborhood cover $\mathcal{C}$. We will ensure that for every regular vertex $v \in V(H)$, the number of clusters in $\mathcal{C}$ to which v ever belongs is at most γ.

Given a regular vertex $v \in V(H)$, we denote by n_v the number of clusters in the current cluster set $\mathcal{C}$ containing v. In order to bound the total number of clusters to which a regular vertex belongs, we use the notion of *vertex budgets* that we define next.

Definition (Vertex Budgets) *Let $x \in V(H)$ be a vertex, and let $C \in \mathcal{C}$ be a cluster that contains x. Then the* budget of x for cluster C *is $\beta_C(x) = \left(1 + \frac{\log W(C)}{\log^2 W}\right) \cdot w(x)$. The total budget of x is $\beta(x) = \sum_{\substack{C\in\mathcal{C}:\\ x\in V(C)}} \beta_C(x)$. For a subset $X \subseteq V(H)$ of vertices, we denote by $\beta(X) = \sum_{x\in X} \beta(x)$.*

Note that for a supernode $u \in U$, $\beta(u) = 0$, since $w(u) = 0$, and for a regular vertex v, for every cluster C containing v, $w(v) \leq \beta_C(v) \leq (1 + 1/\log W)w(v)$, and overall, $w(v)n_v \leq \beta(v) \leq (1 + 1/\log W)w(v)n_v$.

Assume now that we are given some collection $\mathcal{C}$ of clusters of H, and let $C \in \mathcal{C}$ be any such cluster. As graph H undergoes valid update operations, cluster C is updated accordingly. Specifically, consider any update opertion $\sigma_t \in \Sigma$. If σ_t is a deletion of an edge e, and $e \in E(C)$, then we delete e from C as well. If σ_t is a deletion of an isolated vertex x, and $x \in V(C)$, then we delete x from C as well. Assume now that σ_t is a supernode splitting operation, applied to a supernode u, and a set $E' \subseteq \delta_H(u)$ of its incident edges. If $u \notin C$, then we do not need to update the cluster C. Otherwise, we let $E'_C = E' \cap E(C)$. We then update the cluster C by performing a supernode-splitting operation in it, for vertex u, with edge set E'_C. Note that such update operations may not introduce new regular vertices into C. We then immediately obtain the following observation.

Observation 4.1 *If an edge-deletion or a supernode-splitting update operation is performed in graph H, then for every vertex $x \in V(H)$, budget $\beta(x)$ remains unchanged, and for every cluster C containing x, $\beta_C(x)$ remains unchanged. Additionally, for every cluster C, $\sum_{x\in V(C)} \beta_C(x)$ remains unchanged, and the total budget of all vertices $\sum_{x\in V(H)} \beta(x)$ remains unchanged. If an isolated vertex deletion update operation is performed in graph H, with the deleted vertex y, then $\sum_{x\in V(H)} \beta(x)$ decreases by at least $\beta(y)$; for every cluster $C' \in \mathcal{C}$ with $y \notin C'$, the budget $\sum_{x\in V(C)} \beta_C(x)$ does not change, and for every cluster $C' \in \mathcal{C}$ with $y \in C'$, $\sum_{x\in V(C)} \beta_C(x)$ decreases by at least $\beta_C(y)$.*

Recall that $r = 2\left\lceil \frac{\log W}{\log\log W} \right\rceil$. The regular vertices in V are partitioned into classes $S_0, \ldots, S_r$, as follows.

Definition (Vertex Classes) *For a regular vertex $v \in V$ and an integer $0 \leq j < r$, we say that v belongs to class S_j, iff $2^j \leq n_v < 2^{j+1}$. If $n_v \geq 2^r$, then we say that v belongs to class S_r. For all $1 \leq j \leq r$, we denote $S_{\geq j} = S_j \cup \cdots \cup S_r$.*

4.2 Procedure ProcCut

The input to Procedure ProcCut is a cluster $C \in \mathcal{C}$, some vertex $x \in V(C)$, and a distance threshold D (which is identical to the input distance parameter D). The procedure runs Dijkstra's algorithm (or weighted BFS) from vertex x in graph C, up to a certain depth D^*, that will be determined later.

Recall that Dijkstra's algorithm maintains a set S of "discovered" vertices, where at the beginning $S = \{x\}$. Throughout the algorithm, for every vertex $y \in S$, we maintain the distance $\mathsf{dist}_C(x, y)$, and a neighbor vertex a_y of y that does not lie in S, and minimizes the length of the edge (y, a_y). In every iteration, we select a vertex $y \in S$, for which $\mathsf{dist}_C(x, y) + \ell(y, a_y)$ is minimized, and add vertex a_y to S. We are then guaranteed that $\mathsf{dist}_C(x, a_y) = \mathsf{dist}_C(x, y) + \ell(y, a_y)$. Assume that we are given, for every vertex $y \in V(C)$, a list $\lambda(y)$ of its neighbors a, sorted according to the length $\ell(a,y)$ of the corresponding edge, from smallest to largest. Then Dijkstra's algorithm can be implemented so that, if S_i is the set S after the ith iteration, then the total running time of the algorithm up to and including iteration i is $O(E(S_i))$. In order to do so, we maintain, for every vertex $y \in V(G)$, a pointer p_y to the vertex a_y on the list $\lambda(y)$. We also maintain a heap of vertices in set $\{a_y \mid y \in S\}$, whose key is $\mathsf{dist}_G(x,y) + \ell(y,a_y)$. In every iteration, we select a vertex $a = a_y$ from the top of the heap, add it to S, and then advance the pointer p_y until the first vertex that does not lie in S is encountered (if vertex a that was added to S serves as vertex a_y for several vertices $y \in S$, we advance the pointer p_y for each such vertex y). We also initialize pointer p_a.

For all $i \geq 0$, we denote by L_i the set of all vertices of C that lie at distance $2(i-1)D+1$ to $2iD$ from x in C. In other words:

$$L_i = B_C(x, 2iD) \setminus B_C(x, 2(i-1)D).$$

We refer to the vertices of L_i as *layer i of the BFS*. We denote by $W_i = \sum_{y \in L_i} w(y)$ the total weight of all vertices in L_i. Recall that we have denoted by W the total weight of all vertices of H at the beginning of the algorithm. The following definition is crucial for the description of Procedure $\mathsf{ProcCut}$.

Definition (Eligible Layer) *For an integer $i > 1$, we say that layer L_i of the BFS is* eligible *iff the following conditions hold:*

C1. $\sum_{i' \leq i} W_{i'} \leq W(C)/2$;

C2. $W_i \leq \left(\sum_{i' < i} W_{i'}\right) / (64 \log^2 W)$; *and*

C3. for all $1 \leq j \leq r$: $W(L_i \cap S_{\geq j}) \leq \left(\sum_{i'<i} W(L_{i'} \cap S_{\geq j})\right) / \left(64 \log^2 W\right)$.

We need the following claim, whose proof uses standard arguments and appears in Section D.1 of Appendix.

Claim 4.2 *If the total weight of all vertices in $B_C(x, 256D\log^4 W)$ is at most $W(C)/2$, then there is some eligible layer L_i with $1 < i < 128\log^4 W$.*

We are now ready to describe the algorithm for $\mathsf{ProcCut}$. The input to the procedure is a cluster $C \in \mathcal{C}$, a vertex $x \in V(C)$, and a distance threshold $D > 0$. We assume that we are given, for every vertex $y \in V(C)$, a list $\lambda(y)$ of its neighbors a in C, sorted according to the length $\ell(a,y)$ of the corresponding edge, from smallest to largest. The procedure runs Dijkstra's algorithm from the input vertex x in graph C, until it encounters the first layer L_i, such that either $\sum_{i' \leq i} W_{i'} > W(C)/2$, or L_i is an eligible layer. In the former case, the algorithm outputs FAIL. In the latter case, it computes two new clusters: cluster C' is the subgraph of H induced by vertex set $L_1 \cup \cdots \cup L_i$. Cluster C'' is obtained from cluster C by deleting all vertices in $L_1 \cup \cdots L_{i-1}$ from it. The algorithm outputs cluster C' explicitly, by listing all its vertics and edges, and it outputs cluster C'' implicitly, by listing the edges and the vertices of $C \setminus C''$. Notice that vertices of L_i, and edges that connect them, belong to both C' and C''. The following observation immediately follows from Claim 4.2 and the definition of a layer.

Observation 4.3 *If Procedure $\mathsf{ProcCut}(C,x,D)$ does not return FAIL, then it produces two clusters C', C'', with $W(C') \leq W(C)/2$, and $\mathsf{diam}(C') \leq 512D\log^4 W$. Moreover, $V(C') \subseteq B_C(x, 256D\log^4 W)$.*

Next, we establish that the two new clusters C', C'' contain a D-neighborhood every vertex of C in the following claim, whose proof appears in Section D.2 of Appendix.

Claim 4.4 *Assume that Procedure* $\mathsf{ProcCut}(C,x,D)$ *produces two clusters* C' *and* C''*. Then for every vertex* $y \in V(C)$*, either* $B_C(y,D) \subseteq V(C')$ *or* $B_C(y,D) \subseteq V(C'')$ *must hold.*

We view Procedure $\mathsf{ProcCut}$ as creating a new cluster C' that is split off of the cluster C; recall that $W(C') \leq W(C)/2$ must hold. We then update the cluster C, by setting $C = C''$. Notice that the running time of the procedure is $O(|E(C')|) \leq O(W(C')\mu)$.

Lastly, we prove the following lemma, that is central to bounding the number of clusters that a regular vertex may belong to. The proof appears in Appendix D.3 of Appendix.

Lemma 4.5 *Assume that Procedure* $\mathsf{ProcCut}(C,x,D)$*, when applied to a cluster* $C \in \mathcal{C}$*, produced two clusters* C'*,* C''*, and assume that cluster set* $\mathcal{C}$ *was updated accordingly. Denote by* β *and* β' *the total budget of all vertices in* $V(H)$ *at the beginning and at the end of the procedure, respectively. For all* $1 \leq j \leq r$*, let* $\beta_{\geq j}$ *and* $\beta'_{\geq j}$ *denote the total budget of all vertices in* $S_{\geq j}$ *at the beginning and at the end of the procedure, respectively, and let* W'_j *be the total weight of new regular vertices that join class* S_j *at the end of the procedure. Then* $\beta' \leq \beta$*, and moreover, for all* $1 \leq j \leq r$*,* $\beta'_{\geq j} \leq \beta_{\geq j} + 2^j(1+1/\log W)W'_j$*.*

Modified $\mathsf{ProcCut}$**.** So far we have described Procedure $\mathsf{ProcCut}$, whose input is a cluster C, a vertex x, and a distance parameter D. Recall that, if the total weight of all vertices in $B_C(x, 256D\log^4 W)$ is greater than $W(C)/2$, the procedure may return FAIL. We will sometimes use the procedure slightly differently. The input to this modified procedure, that we denote by $\mathsf{ProcCut}'$, is a cluster C, a distance bound D, and two vertices $x, y \in V(C)$ with $\mathsf{dist}_C(x,y) \geq 1024D\log^4 W$. As before, we assume that we are given, for every vertex $a \in V(C)$, a list $\lambda(a)$ of its neigbhors b, sorted by the length of the corresponding edge (a,b). We run the procedure $\mathsf{ProcCut}$ simultaneously from vertex x and from vertex y in graph C. In other words, we run Dijkstra's algorithm from both vertices simultaneously, so that the number of edges that the two algorithms discover at each time step (or, equivalently, the time they invest), remain within a constant factor from each other. Once an eligible layer is reached by either of the procedures, we terminate both algorithms (since $B_C(x, 256D\log^4 W) \cap B_C(y, 256D\log^4 W) = \emptyset$, this is guaranteed to happen, from Claim 4.2, with a layer L_i whose index $1 < i < 128\log^4 W$). Assume w.l.o.g. that $\mathsf{ProcCut}(C,x)$ has reached an eligible layer L_i before $\mathsf{ProcCut}(C,y)$. Let C', C'' be the two resulting clusters, so that $x \in C'$, $V(C') = L_1 \cup \cdots \cup L_i$, while $y \in C''$, with $V(C'') = V(C) \setminus (L_1 \cup \cdots \cup L_{i-1})$, and $W(C') \leq W(C)/2$. We then let C'' be the cluster obtained from C by deleting the vertices of $L_1 \cup \cdots \cup L_{i-1}$. The outcome of the procedure $\mathsf{ProcCut}'(C,x,y,D)$ is the pair C', C'' of clusters, where, as before, cluster C' is returned explicitly, by listing all of its vertices and edges, and cluster C'' is returned implicitly, by listing all vertices and edges of $C \setminus C''$. We summarize the properties of Procedure $\mathsf{ProcCut}'$ in the next claim.

Claim 4.6 *Procedure* $\mathsf{ProcCut}'(C,x,y,D)$*, given a pair* $x, y \in V(C)$ *of vertices with* $\mathsf{dist}_C(x,y) \geq 1024D\log^4 W$ *produces two subgraphs* C', C'' *of* C*, with* $x \in C' \setminus C''$ *and* $y \in C'' \setminus C'$*, or the other way around. Moreover,* $\mathsf{diam}(C') \leq 512D\log^4 W$*,* $W(C') \leq W(C)/2$*, and for every vertex* $z \in V(C)$*, either* $B_C(z,D) \subseteq V(C')$ *or* $B_C(z,D) \subseteq V(C'')$*. The running time of the algorithm is* $O(|E(C')|)$*, provided it is given, for every vertex* $a \in V(C)$*, a list* $\lambda(a)$ *of its neigbhors* b *in* C*, sorted by the length of the corresponding edge* (a,b)*.*

We note that the budget analysis from Lemma 4.5 continues to hold for $\mathsf{ProcCut}'$ as well, as we can equivalently view $\mathsf{ProcCut}'$ as running $\mathsf{ProcCut}$ from one of the vertices, x or y.

4.3 Computing the Initial Neighborhood Cover – Procedure InitNC

We now describe an algorithm, that we refer to as InitNC, for computing an initial neighborhood cover. We assume that we are given a valid input structure $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E}, D\right)$, where the degree of every regular vertex is at most μ, and we denote $D' = 1024D \log^4 W$. Our algorithm will compute a strong (D, D')-neighborhood cover $\mathcal{C}$ for $V(H)$ in H.

The algorithm maintains a cluster $C^* \subseteq H$, where at the beginning, $C^* = H$. For every vertex $y \in V(C^*)$, it maintains a list $\lambda(y)$ of neighbors a of y in C^*, sorted by the length of the corresponding edge (a, y).It also maintains a collection $\mathcal{C}$ of clusters, that is initialized into $\emptyset$. We will ensure that every cluster in $\mathcal{C}$ has diameter at most D'. Lastly, the algorithm maintains a set $U^* \subseteq V(C^*)$ of potential centers, that is initially set to $V(H)$. We initialize the lists $\lambda(y)$ for every vertex $y \in V(H)$ at the beginning. The algorithm performs iterations, as long as $U^* \neq \emptyset$.

In order to perform an iteration, we select an arbitrary vertex $x \in U^*$, and execute ProcCut(C^*, x). Assume first that procedure does not return FAIL, and instead computes two clusters C', C'' (recall that C'' is only returned implicitly, by listing the vertices and edges of $C \setminus C''$). In this case, we say that the iteration is *good*. Note that we are guaranteed, from Observation 4.3, that the diameter of C' is bounded by $D'/2$. We then add C' to $\mathcal{C}$, set $C^* = C''$ (by deleting the edges and vertices of $C \setminus C''$ from C^*), and delete from U^* all vertices that no longer belong to C^*. We also update the lists $\lambda(y)$ of vertices $y \in V(C^*)$ by deleting the neighbors of y that no longer lie in C^*, finishing the current iteration.

Assume now that Procedure ProcCut returned FAIL. In this case, we say that the iteration is *bad*. We are then guaranteed, from Claim 4.2, that the total weight of vertices in $B_{C^*}(x, 256D \log^4 W)$ is at least $W(C^*)/2$. We initialize an ES-tree data structure τ, rooted at vertex x, up to depth $\hat{D} = 512D \log^4 W$. Throughout the remainder of the algorithm, we denote by $B' = B_{C^*}(x, 256D \log^4 W)$, and we denote by $B'' = B_{C^*}(x, 512D \log^4 W)$ – the set of all vertices that belong to τ. Note that both vertex sets can be maintained by the algorithm that maintains the tree τ.

In the remainder of the algorithm, we will always let U^* contain all vertices of $V(C^*) \setminus B''$. Therefore, initially, we delete from U^* all vertices of B''. Whenever some vertex a leaves the set B'', we will add it back to set U^*. We then denote $x^* = x$, and continue to the next iteration. From this time onward, the vertices in B' will never be deleted from C^*, since for every vertex $y \in U^*$, $B' \cap B_{C^*}(y, 256D \log^4 W) = \emptyset$ currently holds, and therefore will continue to hold as vertices and edges are deleted from C^*. Notice that, as vertices and edges are deleted from C^*, over the course of the InitNC algorithm, $W(C^*)$ may decrease, but $W(B')$ will not decrease, and so $W(B') \geq W(C^*)/2$ will continue to hold for the remainder of the algorithm. Therefore, after a single bad iteration, for the remainder of the algorithm, for every vertex $y \in U^*$, $W(B_{C^*}(y, 256D \log^4 W)) \leq W(C^*)/2$ must hold, since $W(B') \geq W(C^*)/2$, and $B' \cap B_{C^*}(y, 256D \log^4 W) = \emptyset$. It follows that at most one iteration of the algorithm may be bad.

The algorithm terminates when $U^* = \emptyset$ holds. If $C^* \neq \emptyset$ at the time of the termination, then $V(C^*) \subseteq B''$ must hold. We then add C^* as the last cluster to $\mathcal{C}$. We call C^* a *distinguished cluster* of $\mathcal{C}$. Note that for every cluster $C \in \mathcal{C} \setminus \{C^*\}$, $W(C) \leq W(H)/2$ holds. It may be sometimes convenient to think of the algorithm InitNC as iteratively splitting the clusters of $\mathcal{C} \setminus \{C^*\}$ off of cluster $C = H$, and so we may view C^* as the updated version of the cluster C at the end of the algorithm, instead of viewing it as a newly created cluster.

From Observation 4.3 and Claim 4.4, it is immediate to verify that the collection $\mathcal{C}$ of clusters that we obtain at the end of the algorithm is a strong (D, D')-neighborhood cover for H.

The running time for a bad iteration is at most $|E(H)|$, while the running time of a good iteration that creates a cluster C' is bounded by $|E(C')|$. Additionally, the running time required for maintaining the ES-tree τ is $O(|E(H)| \cdot D \cdot \operatorname{poly}\log|E(H)|)$. Therefore, the total running time of the algorithm is bounded by $O(|E(H)| \cdot D \cdot \operatorname{poly}\log|E(H)| + \sum_{C\in\mathcal{C}} |E(C)|) \leq O(W(H) \cdot D \cdot \operatorname{poly}\log(W(H)\mu) + \sum_{C\in\mathcal{C}} W(C)) \cdot \mu$. Since, for every vertex $x \in V(H)$ and cluster $C \in \mathcal{C}$ containing x, $\beta_C(x) \geq w(x)$ holds, we get that $\sum_{C\in\mathcal{C}} W(C) \leq \sum_{x\in V(H)} \beta(x)$. Lastly, since, from Lemma 4.5, the total budget of all vertices $\sum_{x\in V(H)} \beta(x)$ does not increase over the course of the algorithm, and since, at the beginning of the algorithm, $\sum_{x\in V(H)} \beta(x) \leq 2W(H)$ held, we get that the total running time is bounded by $O(W(H)\cdot\mu\cdot D\cdot\operatorname{poly}\log(W(H)\mu)) \leq O(N^0(H)\cdot\mu\cdot D\cdot\operatorname{poly}\log(N^0(H)\mu))$, where $N^0(H)$ is the number of regular vertices in H. Therefore, we have proved the following lemma.

Lemma 4.7 *Algorithm* InitNC *described above correctly computes a strong (D, D')-neighborhood cover $\mathcal{C}$ for H, for $D' = 1024D\log^4 W$, where $W = N^0(H)$ is the number of regular vertices in H. The algorithm is deterministic, with running time $O(|E(H)| \cdot D \cdot \operatorname{poly}\log|E(H)| + \sum_{C\in\mathcal{C}} |E(C)|) \leq O(N^0(H) \cdot D \cdot \mu \cdot \operatorname{poly}\log(N^0(H)\mu))$.*

Note that Procedure InitNC can be viewed as a series of applications of Procedure ProcCut. We will sometimes use this view, which will, for example, allow us to use the results from Lemma 4.5 to bound the budgets of vertices.

4.4 Algorithmic Framework for Maintaining Neighborhood Cover

We now describe a general algorithmic framework for maintaining a strong neighborhood cover in a given valid input structure that undergoes a sequence of valid update operations. This framework will be used in the proof of Theorem 3.3 in several different ways. We denote the algorithm described in this subsection by AlgMaintainNC.

We assume that we are given a valid input structure $\mathcal{I} = \left(H, \{\ell(e)\}_{e\in E(H)}, D\right)$, that undergoes a sequence of valid update operations, with dynamic degree bound μ. The algorithm maintains a collection $\mathcal{C}$ of subgraphs of H that we refer as clusters, with the guarantee that for every regular vertex $v \in V(H)$, there is a cluster $C \in \mathcal{C}$ with $B_H(v, D) \subseteq V(C)$. The algorithm maintains, for every regular vertex $v \in V(H)$, a cluster $C = \text{CoveringCluster}(v)$, with $B_H(v, D) \subseteq V(C)$. It also maintains, for every vertex $x \in V(H)$, a list $\text{ClusterList}(x)$ of all clusters in $\mathcal{C}$ containing x, and for every edge $e \in E(H)$, a list $\text{ClusterList}(e)$ of all clusters in $\mathcal{C}$ containing e. Lastly, for every cluster $C \in \mathcal{C}$ and vertex $x \in V(C)$, it maintains a list $\lambda_C(x)$ of all neighbors a of x in C, sorted by the length of the corresponding edge (a, x).

As before, we set the weight of every regular vertex to be 1, and of every supernode to be 0. We denote by $N^0(H)$ the number of regular vertices in H at the beginning of the algorithm, and by $W = N^0(H)$ the weight of all vertices of H at the beginning of the algorithm.

At the beginning of the algorithm, before any update operations from Σ are processed, we apply Procedure InitNC to graph H, and we add to $\mathcal{C}$ the resulting collection of clusters, that form a strong $(D, 1024D\log^4 W)$-neighborhood cover of H. We also initialize all data structures $\text{CoveringCluster}(v)$ for regular vertices $v \in V(H)$, $\text{ClusterList}(x)$ for vertices $x \in V(H)$, and $\text{ClusterList}(e)$ for edges $e \in E(H)$. For every cluster $C \in \mathcal{C}$ and vertex $x \in V(C)$, we initialize the list $\lambda_C(x)$ of neighbors of x in C.

As graph H undergoes a sequence Σ of valid update operations, we update every cluster $C \in \mathcal{C}$ accordingly. Specifically, consider any update operation $\sigma_t \in \Sigma$. If σ_t is a deletion of an edge e, then for every cluster $C \in \text{ClusterList}(e)$, we delete e from C as well. If σ_t is a deletion of an isolated vertex x, then for every cluster $C \in \text{ClusterList}(x)$, we delete x from C as well. Assume now that σ_t is a supernode splitting operation, applied to a supernode u, and a set $E' \subseteq \delta_H(u)$ of its incident edges.

For every cluster $C \in \mathsf{ClusterList}(u)$, we let $E'_C = E' \cap E(C)$. If $E'_C \neq \emptyset$, then we update the cluster C by performing a supernode-splitting operation in it, for vertex u, with edge set E'_C. We then add C to $\mathsf{ClusterList}(u')$ of the newly created supernode u', and also initialize $\mathsf{ClusterList}(e)$ for every edge e that was just added to H. In order to implement these updates efficiently, we process the edges of E' one-by-one. Let $e = (u, v)$ be an edge of E' that is currently processed. We then consider every cluster $C \in \mathsf{ClusterList}(e)$ one-by-one. For each such cluster C, we mark C as a cluster on which the supernode splitting operation needs to be executed (if it has not been marked yet), and add the edge e to the set E'_C (if set E'_C is not yet initialized, we set $E'_C = \{e\}$). Once all edges in E' are processed, we perform a supernode splitting operation on each marked cluster C, using the edge set E'_C. After each update operation, we update the lists $\lambda_C(x)$ of all relevant vertices $x \in V(C)$.

Note that the total processing time of the update operation σ_t is asymptotically bounded by the total number of edges deleted from the clusters in $\mathcal{C}$, or inserted into the clusters in $\mathcal{C}$, and the total number of vertices deleted from the clusters in $\mathcal{C}$, while processing σ_t.

Additionally, we assume that we are given an oracle, that, at any time, may raise a flag F_C for a cluster $C \in \mathcal{C}$. When the oracle raises flag F_C, it needs to provide a pair $x, y \in V(C)$ of vertices of C with $\mathsf{dist}_C(x, y) \geq 1024D \log^4 W$. The algorithm then runs $\mathsf{ProcCut}'(C, x, y, D)$, obtaining two clusters C', C''. The procedure outputs cluster C' explicitly, and it outputs cluster C'' implicitly, by listing the edges and the vertices of $C \setminus C''$. Assume w.l.o.g. that Procedure $\mathsf{ProcCut}'$ found an eligible layer L_i when running Dijktra's algorithm from vertex x. Then for every regular vertex $v \in V(H)$ with $\mathsf{CoveringCluster}(v) = C$, if $v \in B_C(x, 2iD - D)$, we set $\mathsf{CoveringCluster}(v) = C'$; otherwise, we are guaranteed that $B_H(v, D) \subseteq V(C'')$, and $\mathsf{CoveringCluster}(v)$ does not need to be updated. Additionally, for every vertex $x \in V(C')$, and for every edge $e \in E(C')$, we add C' to $\mathsf{ClusterList}(x)$ or to $\mathsf{ClusterList}(e)$, respectively. Similarly, for every edge e that was deleted from C, and for every vertex x that was deleted from C, we delete C from $\mathsf{ClusterList}(x)$ or from $\mathsf{ClusterList}(e)$, respectively. For every vertex $x \in V(C')$, we initialize the neighbor list $\lambda_{C'}(x)$, and for every edge e that was deleted from C, we update the neighbor lists $\lambda(x)$ of its endpoint(s) that remain in C. Note that all these updates can be made in time $O(|E(C')|)$. We say that cluster C' was *split off of* C. Once the update is completed, flag F_C is lowered. No update operations from Σ are processed when flag F_C is up for any cluster C.

This completes the description of Algorithm $\mathsf{AlgMaintainNC}$. Note that for each cluster $C \in \mathcal{C}$, from the moment C is added to $\mathcal{C}$, it undergoes a sequence of updates that are valid update operations for the corresponding valid input structure $\mathcal{I}_C = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$. Some of these update operations mirror the update operations from Σ that graph H undergoes, and some update operations (edge deletions and isolated vertex deletions) arise from splitting clusters off of C. Next, we analyze some properties of $\mathsf{AlgMaintainNC}$ that will be useful for us later.

Bounding the running time. Recall that we denote by $W = W(H^0)$ the total weight of all vertices of H at the beginning of the algorithm. For every cluster $C \in \mathcal{C}$, we denote by $W^0(C)$ the total weight of all vertices of C at the time when C is added to $\mathcal{C}$. We also denote by $\beta^0(C)$ the sum of the budgets $\beta_C(v)$ of all vertices $v \in V(C)$ at the moment when cluster C is created. From the definition of budgets, $W^0(C) \leq \beta^0(C)$ for every cluster C. Recall that, from the time that C is added to $\mathcal{C}$, $W(C)$ may only decrease. Notice that, once C is added to $\mathcal{C}$, it remains there until the end of the algorithm, though it is possible that all edges and all vertices are deleted from C, and it becomes empty. We denote by $\mathcal{C}'$ the set of all clusters that ever belonged to $\mathcal{C}$ (or equivalently, it is the set of all clusters lying in $\mathcal{C}$ at the end of the algorithm.) We start with the following claim.

Claim 4.8 $\sum_{C \in \mathcal{C}'} W^0(C) \leq O(W \log W)$.

Proof: From the above discussion, for every cluster $C \in \mathcal{C}'$, $W^0(C) \leq \beta^0(C)$. Moreover, it is easy to see that, if we denote by β^0 the sum of the budgets of all vertices $x \in V(H)$ at the beginning of the algorithm, then $\beta^0 \leq 2W$. Therefore, it is enough to prove that $\sum_{C \in \mathcal{C}'} \beta^0(C) \leq O(\beta^0 \log W)$.

We construct a partitioning tree τ, whose vertex set is $\{v(C) \mid C \in \mathcal{C}'\}$. The root of the tree is the vertex $v(H)$, corresponding to the original graph H. Consider now some vertex $v(C)$ of the tree, where C is some cluster. Let $C_1, C_2, \ldots, C_q$ be all clusters that were split off of C over the course of the algorithm. Recall that, from Condition C1, we are guaranteed that for all $1 \leq i \leq q$, $W(C_i) \leq W(C)/2$ holds when C_i is split off of C, and so $W^0(C_i) \leq W^0(C)/2$. We add edges connecting each of the vertices $v(C_1), \ldots, v(C_q)$ to $v(C)$, and these vertices become children of the vertex $v(C)$ in the tree. We need the following observation.

Observation 4.9 *Let $v(C)$ be a vertex in the tree τ, and let $v(C_1), \ldots, v(C_q)$ be its child vertices. Then $\sum_{i=1}^{q} \beta^0(C_i) \leq \beta^0(C)$.*

Assume first that the observation is correct. Since, for every child vertex $v(C_i)$ of vertex $v(C)$, $W^0(C_i) \leq W(C)/2 \leq W^0(C)/2$ holds, the depth of the tree τ is bounded by $\lceil \log W \rceil$. Moreover, if we denote, for $1 \leq i \leq \lceil \log W \rceil$, by $\mathcal{C}_i \subseteq \mathcal{C}'$ the set of all clusters C such that the distance from $v(C)$ to the root of τ is exactly i, then, from Observation 4.9, $\sum_{C \in \mathcal{C}_i} \beta^0(C) \leq \beta^0(H)$. Therefore, $\sum_{C \in \mathcal{C}'} W^0(C) \leq \sum_{C \in \mathcal{C}'} \beta^0(C) \leq O(\beta^0(H) \log W) \leq O(W \log W)$. In order to complete the proof of Claim 4.8, it is now enough to prove Observation 4.9.

Proof of Observation 4.9. We assume that the clusters $C_1, C_2, \ldots, C_q$ where split off of C in this order. Consider the iteration of the algorithm when cluster C_i was split off of cluster C, by applying Procedure ProcCut to cluster C. Let C'' denote the cluster C at the end of this procedure, and let C denote the same cluster at the beginning of this procedure. From Lemma 4.5, the total budget of all vertices in H did not increase as the result of applying ProcCut to cluster C. If we denote β the sum of budgets of all vertices of H before the procedure is applied to cluster C, and by β' the sum of budgets of all vertices after the procedure is applied, then:

$$\beta' - \beta = \sum_{v \in C_i} \beta_{C_i}(v) + \sum_{v \in C''} \beta_{C''}(v) - \sum_{v \in C} \beta_C(v).$$

Since $\beta' \leq \beta$, we get that $\sum_{v \in C_i} \beta_{C_i}(v) + \sum_{v \in C''} \beta_{C''}(v) \leq \sum_{v \in C} \beta_C(v)$. By applying this argument iteratively to all clusters $C_1, \ldots, C_q$, and recalling that, from Observation 4.1, update operation cannot increase the total budget of vertices in a cluster, we get that $\sum_{i=1}^{q} \beta^0(C_i) \leq \beta^0(C)$. □ □

We obtain the following immediate corollary of Claim 4.8, bounding the running time of Algorithm AlgMaintainNC.

Corollary 4.10 *The running time of* AlgMaintainNC *is $O(W \cdot \mu \cdot D \cdot \operatorname{poly}\log(W\mu))$.*

Proof: Recall that, from Lemma 4.7, the running time of Algorithm InitNC is $O(W\mu D \operatorname{poly}\log(W\mu))$. For every cluster $C \in \mathcal{C}$, let $m(C)$ be the total number of edges that ever belonged to cluster C over the course of the algorithm. Clearly, $m(C) \leq W^0(C) \cdot \mu$. The time spent on creating the cluster C for the first time (by splitting it off of some other cluster C^*, including the time needed to update C^*) is bounded by $O(m(C))$. Subsequently, the time needed to process all update operations on C is bounded by the total number of edges that are either deleted from C or inserted into C, and the total number of vertices deleted from C. As the number of supernodes that are ever present in C is bounded by $m(C)$, we get that the total update time spent on processing all update operations is bounded by $O(m(C))$. The total running time of the algorithm, excluding the time needed to run InitNC, is then bounded by: $\sum_{C \in \mathcal{C}'} O(m(C)) \leq \sum_{C \in \mathcal{C}'} O(W^0(C) \cdot \mu) \leq O(W\mu \log W)$. □

Bounding number of copies of each vertex. Next, we bound the number of clusters in $\mathcal{C}'$ that may contain a regular vertex v of H, in the following theorem.

Theorem 4.11 *For every regular vertex $v \in V(H)$, the total number of clusters $C \in \mathcal{C}'$, such that v lied in C at any time during the algorithm's execution is at most $W^{O(1/\log\log W)}$.*

Proof: For the sake of the proof, it is convenient to disregard update operations when isolated vertices are deleted from H; we will simply assume that such vertices remain in graph H and in every cluster C to which they belonged at the time of deletion; we will not add such vertices to any new cluster. This is done in order to avoid the reduction in $W(H)$ (and in the budget of all vertices) following isolated vertex deletion.

Algorithm AlgMaintainNC can be equivalently described as follows. We start with $\mathcal{C} = \{H\}$, and then perform iterations. In ever iteration, we apply Procedure ProcCut to a cluster $C \in \mathcal{C}$, with a vertex $x \in V(C)$ and distance bound D, obtaining two clusters C', C''. We then add C' to $\mathcal{C}$ and replace C with C'' in $\mathcal{C}$. (We also perform edge-deletion and supernode-splitting operations, but these operations do not affect vertex weights or budgets so they are immaterial to this discussion).

Consider now some application of the ProcCut operation to some cluster $C \in \mathcal{C}$, in which a new cluster C' is split off of C; we denote by C'' the cluster C at the end of this operation. Recall that $W^0(C') \leq W(C)/2 \leq W^0(C)/2$ must hold. If a regular vertex $v \in V(C)$ lies in C' at the end of this operation, but it does not lie in C'', then we say that this copy of v was *moved from cluster C to cluster C'*. If vertex v lies in both C' and C'', then we say that *a new copy of v was created* – the copy that lies in C'. Since, whenever vertex v is moved from cluster C to cluster C', $W^0(C') \leq W^0(C)/2$ must hold, a copy of v may only be moved from one cluster to another at most $O(\log W)$ times. Therefore, it is now enough to bound the total number of copies of a vertex that the algorithm may ever create. Since we do not delete isolated vertices, once a new copy of a vertex v is created, it is never deleted, and may only move from cluster to cluster. In particular, if a regular vertex v is added to set $S_{\geq j}$, then it remains in set $S_{\geq j}$ until the end of the algorithm. Recall that vertex v belongs to class S_r iff at least 2^r copies of vertex v exist in $\mathcal{C}$. We show in the next claim that at the end of the algorithm, $S_r = \emptyset$. It then follows that for any vertex v that every belonged to the graph H, the total number of copies of v that may be created over the course of the algorithm is bounded by 2^r. Since a copy of v may be moved at most $O(\log W)$ times, the total number of clusters in $\mathcal{C}'$ to which v ever belonged over the course of the algorithm is bounded by $O(2^r \log W) \leq W^{O(1/\log\log W)}$, since $r = 2\lceil \log W/\log\log W \rceil$. The following claim will then finish the proof of Theorem 4.11.

Claim 4.12 *Throughout the algorithm, $S_r = \emptyset$ holds.*

Proof: We prove by induction that, for all $1 \leq j \leq r$, $W(S_{\geq j}) \leq W/\log^j W$ holds over the course of the entire algorithm. Assume first that this is indeed the case. Then we get that $W(S_r) \leq W/\log^r W < 1$, since $r = 2\lceil \log W/\log\log W \rceil$. Since the weight of every regular vertex is 1, it follows that $S_r = \emptyset$. Therefore, it is now enough to prove the following claim.

Claim 4.13 *For all $1 \leq j \leq r$, $W(S_{\geq j}) \leq W/\log^j W$ holds over the course of the entire algorithm.*

Proof: Consider some index $1 \leq j \leq r$, and some regular vertex $v \in V(H)$. Vertex v may only be added to $S_{\geq j}$, if $v \in S_{j-1}$, and Procedure ProcCut creates an additional copy of vertex v, so that the number of copies of v becomes 2^j. From the moment a vertex is added to $S_{\geq j}$, it remains there until the end of the algorithm. Therefore, it is enough to prove that, at the end of the algorithm, $W(S_{\geq j}) \leq W/\log^j W$ holds.

We prove this by induction on j. We start with the base case, where $j = 1$. Notice that, from the definition of vertex budgets, for every regular vertex $v \in V \setminus S_{\geq 1}$, $w(v) \leq \beta(v) \leq w(v)(1 + 1/\log W)$ holds throughout the algorithm. At the beginning algorithm, the total budget of all regular vertices, $\beta = \sum_{v\in V} \beta(v) \leq \sum_{v\in V} w(v)(1 + 1/\log W) = W(1 + 1/\log W)$ holds (recall that supernodes all have weight 0 and budget 0). Since we assume that isolated vertices are not deleted from H, throughout the algorithm, $\sum_{v\in V} w(v) = W$ holds. Assume for contradiction that $\sum_{v\in S_{\geq 1}} w(v) > W/\log W$ holds at the end of the algorithm. Observe that, at the end of the algorithm, a regular vertex $v \in V \setminus S_{\geq 1}$ has budget $\beta(v) \geq w(v)$, while a vertex $v \in S_{\geq 1}$ has budget at least $2w(v)$. Therefore, the total budget of all vertices at the end of the algorithm is at least:

$$\sum_{v\in V} w(v) + \sum_{v\in S_{\geq 1}} w(v) > W(1 + 1/\log W).$$

This is a contradiction since the total budget of all vertices may not increase over the course of the algorithm.

Assume now that for some $1 \leq j < r$, $\sum_{v\in S_{\geq j}} w(v) \leq W/\log^j W$ holds at the end of the algorithm. We will now prove that this inequality holds for $j + 1$.

Note that every vertex v that belongs to $S_{\geq j}$ at the end of the algorithm has budget at least $2^j w(v)$, while a vertex v that belongs to $S_{\geq j+1}$ has budget at least $2^{j+1} w(v)$ at the end of the algorithm. Assume for contradiction that at the end of the algorithm, $\sum_{v\in S_{\geq j+1}} w(v) > W/\log^{j+1} W$ holds. Then, at the end of the algorithm:

$$\sum_{v\in S_{\geq j}} \beta(v) \geq 2^j \sum_{v\in S_{\geq j}} w(v) + 2^j \sum_{v\in S_{\geq j+1}} w(v) > 2^j \sum_{v\in S_{\geq j}} w(v) + 2^j W/\log^{j+1} W. \tag{1}$$

Consider now changes to the value $\beta_{\geq j} = \sum_{v\in S_{\geq j}} \beta(v)$ over the course of the algorithm. When the algorithm starts, $\beta_{\geq j} = 0$. From Lemma 4.5, the value $\beta_{\geq j}$ may only increase when a vertex v is added to $S_{\geq j}$. In such a case, the increase in $\beta_{\geq j}$ is bounded by $2^j(1 + 1/\log W)w(v)$. Update operations in the input sequence Σ and other updates due to application of ProcCut may not increase $\beta_{\geq j}$. Therefore, at the end of the algorithm:

$$\begin{aligned}
\beta_{\geq j} &\leq 2^j(1 + 1/\log W) \cdot \sum_{v\in S_{\geq j}} w(v) \\
&\leq 2^j \sum_{v\in S_{\geq j}} w(v) + 2^j \sum_{v\in S_{\geq j}} w(v)/\log W \\
&\leq 2^j \sum_{v\in S_{\geq j}} w(v) + 2^j W/\log^{j+1} W,
\end{aligned}$$

from the induction hypothesis, contradicting Inequality 1. □ □ □

4.5 MaintainCluster Problem and a Reduction from RecDynNC to MaintainCluster

In this subsection, we define a new problem, that we call MaintainCluster. Intuitively, this problem will be used in order to simulate the oracle needed for AlgMaintainNC. Additionally, an algorithm for this problem will need to support short-path-query(C, v, v') queries as required in the RecDynNC problem. We state our main result for the MaintainCluster problem – a fast deterministic algorithm for solving this problem, and show that this new algorithm, combined with Algorithm AlgMaintainNC presented in the previous subsection, provides an algorithm for the RecDynNC problem, leading to the proof of Theorem 3.3. We start with the definition of the MaintainCluster problem.

4.5.1 MaintainCluster **Problem**

Intuitively, the input to the MaintainCluster problem is a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e\in E(C)}, D\right)$, where C is a connected subgraph of the original graph H. Graph C undergoes a sequence of valid update operations, with dynamic degree bound μ. We assume that we are given as input a parameter $\hat{W} \geq N^0(C)\mu$, where $N^0(C)$ is the initial number of regular vertices in C. We are required to support queries short-path-query(C, v, v'), in which, given a pair $v, v' \in V(C)$ of regular vertices of C, we need to return a path of length at most αD connecting them in C, where α is the desired approximation factor. However, the algorithm is allowed to raise a flag F_C at any time. When flag F_C is raised, the algorithm is required to supply a pair v, v' of regular vertices of C, with $\mathsf{dist}_C(v, v') > 1024D\log^4\hat{W}$. The algorithm then receives, as part of its input update sequence Σ, a sequence Σ' of valid update operations, at the end of which either x or y are deleted from C. We called sequence Σ' a *flag lowering sequence*. Once the flag lowering sequence Σ' is processed, flag F_C is lowered. However, the algorithm may raise the flag F_C again immediately, as long as it provides a new pair $\hat{v}, \hat{v}'$ of vertices with $\mathsf{dist}_C(\hat{v}, \hat{v}') > 1024D\log^4\hat{W}$. We emphasize that we view the resulting flag lowering sequences Σ' as part of the input sequence Σ of valid update operations, and so the restriction that the total number of edges incident to a regular vertex of C over the course of the update sequence is bounded by μ continues to hold. Queries short-path-query may only be asked when flag F_C is down. We also emphasize that the initial cluster C that serves as input to the MaintainCluster problem may have an arbitrarily large diameter, and so the algorithm for the MaintainCluster problem may repeatedly raise the flag F_C, until it is able to support short-path-query(C, v, v') (that intuitively means that the diameter of C has fallen under αD). We now provide a formal definition of the problem.

Definition (MaintainCluster problem) *The input to the* MaintainCluster *problem is a valid input structure* $\mathcal{I} = \left(C, \{\ell(e)\}_{e\in E(C)}, D\right)$*, where* C *is a connected graph. Graph* C *undergoes an online sequence* Σ *of valid update operations, and we are given its dynamic degree bound* μ*. Additionally, we are given the desired approximation factor* α *and a parameter* $\hat{W} \geq N^0(C)\cdot\mu$*, where* $N^0(C)$ *is the number of regular vertices of* C *at the beginning of the algorithm. The algorithm must support queries* short-path-query(C, v, v')*: given a pair* $v, v' \in V(C)$ *of regular vertices of* C*, return a path* P *of length at most* αD *connecting them in* C*, in time* $O(|E(P)|)$*. The algorithm may, at any time, raise a flag* F_C*, at which time it must supply a pair* $\hat{v}, \hat{v}'$ *of regular vertices of* C*, with* $\mathsf{dist}_C(\hat{v}, \hat{v}') > 1024D\log^4\hat{W}$*. Once flag* F_C *is raised, the algorithm will obtain, as part of its input update sequence* Σ*, a sequence* Σ' *of valid update operations called flag-lowering sequence, at the end of which either* $\hat{v}$ *or* $\hat{v}'$ *are deleted from* C*. Flag* F_C *is lowered after these updates are processed by the algorithm. Queries* short-path-query *may only be asked when flag* F_C *is down.*

Our main result for the MaintainCluster problem is summarized in the following theorem.

Theorem 4.14 *There is a deterministic algorithm for the* MaintainCluster *problem, that, on input* $\mathcal{I} = \left(C, \{\ell(e)\}_{e\in E(C)}, D\right)$*, that undergoes a sequence of valid update operations with dynamic degree bound* μ*, and parameters* $c/\log\log N^0(C) < \epsilon < 1$ *for some large enough constant* c*, and* $\hat{W} \geq N^0(C)\mu$*, where* $N^0(C)$ *is the number of regular vertices in* C *at the beginning of the algorithm, achieves approximation factor* $\alpha = (\log\hat{W})^{2^{O(1/\epsilon)}}$*, and has total update time* $O\left(N^0(C)\cdot\mu^2\cdot D^3\cdot\hat{W}^{O(\epsilon)}\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right)$.

Next, we show that Theorem 3.3 follows from Theorem 4.14.

4.5.2 Completing the Proof of Theorem 3.3

Recall that we are given a valid input structure $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E(H)}, D\right)$, undergoing a sequence Σ of valid update operations with dynamic degree bound μ, and a desired approximation factor α. As before, we set the weight of every regular vertex to 1, the weight of every supernode to 0, and we denote by $W = N^0(H)$ the weight of all vertices of H at the beginning of the algorithm. We are also given a precision parameter $c/\log\log W < \epsilon < 1$, and we use a new parameter $\hat{W} = W \cdot \mu$.

We employ Algorithm AlgMaintainNC from Section 4.4 on the input $\mathcal{I}$, and the sequence Σ of valid update operations. Recall that the algorithm maintains a set $\mathcal{C}$ of clusters. Whenever Algorithm AlgMaintainNC adds a new cluster C to $\mathcal{C}$, we initialize the algorithm for the MaintainCluster problem from Theorem 4.14 on cluster C, with the same approximation factor α, distance bound D, and parameter $\hat{W}$ as defined above. We denote by $N^0(C) = W^0(C)$ the number of regular vertices of C (or equivalently, the total weight of all vertices of C), when C is added to $\mathcal{C}$. This algorithm will serve two purposes: first, it will process queries short-path-query(C, v, v') for cluster C. Additionally, it will serve as an oracle that is used by Algorithm AlgMaintainNC in order to update the clusters.

Recall that Algorithm AlgMaintainNC starts with $\mathcal{C} = \{H\}$. The only changes that Algorithm AlgMaintainNC may perform to the clusters of $\mathcal{C}$ are the following:

- delete an edge from C (when the corresponding edge is deleted from H);
- delete an isolated vertex from C (when the corresponding vertex is deleted from H);
- add a new supernode u' together with adjacent edges when a supernode $u \in C$ undergoes supernode splitting update in H; and
- split a cluster C' off of C.

(We view the initial execution of the Algorithm InitNC on the original graph H as a series of cluster splitting operations). In the last operation, cluster C' is a vertex-induced subgraph of C. The operation can be implemented by first adding C' to $\mathcal{C}$, and then deleting edges and vertices from C as necessary. Therefore, if we denote by $\mathcal{U} = \{V(C) \mid C \in \mathcal{C}\}$, then set $\mathcal{U}$ only undergoes allowed changes. Algorithm AlgMaintainNC takes care of maintaining the data structures CoveringCluster(v) for every regular vertex $v \in V(H)$, and data structures ClusterList(x) for vertices $x \in V(H)$ and ClusterList(e) for edges $e \in E(H)$. In order to respond to a query short-path-query(C, v, v'), we simply run this query in the data structure for the MaintainCluster problem on cluster C, which must return a path P of length at most αD connecting v to v' in C, in time $O(|E(P)|)$. The fact that the data structures for the MaintainCluster problem on the clusters in $\mathcal{C}$ can support such queries proves that at any time, $\mathcal{C}$ is a strong $(D, \alpha \cdot D)$-neighborhood cover for the regular vertices in graph H.

Note that, from Theorem 4.11, for every regular vertex $v \in V(H)$, the total number of clusters in $\mathcal{C}$ to which vertex v ever belonged over the course of the algorithm is bounded by $W^{O(1/\log\log W)}$.

It now remains to bound the total running time of the algorithm. From Corollary 4.10, the running time of AlgMaintainNC is $O(W \cdot D \cdot \mu \cdot \operatorname{poly}\log(W\mu))$.

For every cluster $C \in \mathcal{C}$, the running time of the algorithm for the MaintainCluster problem on C is at most $O\left(W^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$. Lastly, if we denote by $\mathcal{C}'$ the set of all clusters that were ever added to $\mathcal{C}$, then, from Claim 4.8, $\sum_{C \in \mathcal{C}'} W^0(C) \le O(W \log W)$. Therefore, the total running time of algorithms MaintainCluster(C) for all clusters $C \in \mathcal{C}$ is bounded by:

$$O\left(\left(\sum_{C\in\mathcal{C}'} W^0(C)\right)\cdot\mu^2\cdot D^3\cdot \hat{W}^{O(\epsilon)}\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right)$$
$$\le O\left((N^0(H))^{1+O(\epsilon)}\cdot\mu^{2+O(\epsilon)}\cdot D^3\cdot(\log(N^0(H)\mu))^{O(1/\epsilon^2)}\right),$$

as required.

The remainder of this paper focuses on the proof of Theorem 4.14. As we have shown, the proof of Theorem 3.3, and hence of Theorem 3.4, and Theorems 1.1 and 1.2 follow from it. We start with a slow and simple algorithm for the MaintainCluster problem, that will be used by our final recursive algorithm for the MaintainCluster problem as a recursion base.

4.6 A Slow and Simple Algorithm for the MaintainCluster Problem

In this subsection, we present a straightforward algorithm, that we refer to as AlgSlow, for the MaintainCluster problem. Recall that in this problem, we are given a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e\in E(C)}, D\right)$, where C is a connected graph that undergoes a sequence Σ of valid update operations with dynamic degree bound μ. We denote by $N^0(C)$ the number of regular vertices in C at the beginning of the algorithm. We are also given a parameter $\hat{W}\ge N^0(C)\cdot\mu$. The goal is to to support queries short-path-query(C,v,v'): given a pair $v,v'\in V(C)$ of regular vertices of C, return a path P of length at most αD connecting them in C, in time $O(|E(P)|)$, where α is the desired approximation factor. The algorithm may raise a flag F_C at any time; it is then required to provide supply a pair $\hat{v},\hat{v}'$ of regular vertices of C, with $\mathsf{dist}_C(\hat{v},\hat{v}') > 1024D\log^4\hat{W}$. Once flag F_C is raised, the algorithm will obtain, as part of its input update sequence, a sequence Σ' of update operations, at the end of which either $\hat{v}$ or $\hat{v}'$ are deleted from C. Flag F_C is lowered after these updates are processed by the algorithm. Queries short-path-query may only be asked when flag F_C is down. Our slow and simple algorithm for the MaintainCluster problem is summarized in the following theorem.

Theorem 4.15 *There is a deterministic algorithm for the* MaintainCluster *problem, that we call* AlgSlow*, that achieves approximation factor $\alpha = 1024\log^4\hat{W}$ and total update time:*

$$O((N^0(C))^2\mu D\operatorname{poly}\log\hat{W}).$$

Proof: The algorithm simply maintains, for every regular vertex $v\in V(C)$, the generalized ES-Tree data structure $\tau(v)$ rooted at v, with depth threshold αD, using the algorithm from Theorem 3.2. For each such tree $\tau(v)$, it also maintains a list $L(v)$ of all regular vertices of C that do not lie in the tree $\tau(v)$. Lastly, for every vertex $x\in V(C)$, we maintain a pointer from x to its location in each of the trees $\tau(v)$ that contain x, and similarly for every edge $e\in E(C)$, we maintain a pointer from e to its location in each of the trees $\tau(v)$ containing e. It is immediate to generalize the algorithm from Theorem 3.2 to maintain these additional data structures.

Since the running time of the algorithm from Theorem 3.2 is $\widetilde{O}(N^0(C)\cdot\mu\cdot\alpha D) = \widetilde{O}(N^0(C)\cdot\mu\cdot D\log^4\hat{W})$, the total update time needed to maintain all these data structures is $O((N^0(C))^2\cdot\mu\cdot D\cdot\operatorname{poly}\log\hat{W})$.

Whenever, for any regular vertex v, $L(v)\neq\emptyset$ holds, we raise the flag F_C, with the corresponding pair of vertices being v,v', where v' is any vertex of $L(v)$. We are then guaranteed that $\mathsf{dist}_C(v,v') > 1024D\log^4\hat{W}$.

(We note that it is possible that the diameter of the initial cluster C is greater than αD. In this case, immediately after initializing our data structures, we will discover that for some regular vertices v,

$L(v) \neq \emptyset$ holds. We will then raise the flag F_C immediately, and will continue raising it until $L(v) = \emptyset$ holds for all regular vertices v, and so the diameter of C falls below αD. Only then do we start processing the updates from the input sequence and responding to queries).

Recall that a query short-path-query(C, v, v') may only be asked when flag F_C is down, so for every regular vertex $v'' \in V(C)$, $L(v'') = \emptyset$. In particular, $v' \in \tau(v)$ must hold. Given such a query, we retrace the path P connecting v' to v in the tree $\tau(v)$, and then return this path. Clearly, the running time of this algorithm is $O(|E(P)|)$. Since the depth of the tree $\tau(v)$ is αD, the length of path P is at most αD, as required. $\square$

We will use AlgSlow as a subroutine in order to solve the MaintainCluster problem on clusters that are small, e.g. clusters C with $W^0(C) < \hat{W}^{O(\epsilon)}$. Additionally, for some settings of the parameters, Algorithm AlgSlow immediately provides the desired running time. We summarize these settings in the following observation that follows immediately from Theorem 4.15.

Observation 4.16 *Consider an input to the* MaintainCluster *problem, consisting of a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$ that undergoes an online sequence Σ of valid update operations with dynamic degree bound μ, and a parameter $\hat{W} \geq N^0(C) \cdot \mu$. Then:*

- *if $D > \hat{W}$, then the running time of* AlgSlow *is $O(N^0(C)\mu D^2 \operatorname{poly}\log \hat{W})$;*
- *if $\hat{W} < 2^{O(1/\epsilon^2)}$, then the running time of* AlgSlow *is $O(N^0(C)\mu D \cdot 2^{O(1/\epsilon^2)} \operatorname{poly}\log \hat{W}) \leq O\left(N^0(C)\mu D (\log \hat{W})^{O(1/\epsilon^2)}\right)$.*

5 Second Main Tool: Balanced Pseudocuts, Expanders, and Their Embeddings

Throughout this section, we assume that we are given a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, where C is a connected graph, with weights $w(v) \geq 1$ on regular vertices $v \in V(C)$ and $w(u) = 0$ on supernodes $u \in V(C)$. In this section we develop some tools that will be used in order to design a more efficient algorithm for the MaintainCluster problem.

5.1 Balanced Pseudocuts

Intuitively, following a rather standard definition, for a given balance parameter ρ, a *balanced multicut* in a vertex-weighted graph C is a subset $E' \subseteq E(C)$ of edges of C, such that every connected component of $C \setminus E'$ has weight at most $W(C)/\rho$. We use instead a notion of *balanced pseudocuts*, that can be viewed as a relaxation of the notion of a balanced multicut. The advantage of using this relaxed notion is that, as we later show, we can obtain an algorithm that computes a balanced pseudocut, and embeds an expander X, whose vertex set corresponds to a large subset of the edges of the pseudocut, into the graph C with low congestion. We start by formally defining balanced pseudocuts (that we call pseudocuts for brevity).

Definition (Pseudocut) *Suppose we are given a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$ with weights $w(x) \geq 0$ for vertices $x \in V(C)$, together with another distance bound $\hat{D} > D$ and a parameter $\rho > 1$. A subset $\hat{E} \subseteq E(C)$ of edges of C is a $(\hat{D}, \rho)$-*pseudocut*, iff for every vertex $x \in V(C)$, the total weight of all vertices in $B_{C \setminus \hat{E}}(x, \hat{D})$ is at most $\frac{W(C)}{\rho}$.*

One useful feature of a $(\hat{D}, \rho)$-pseudocut $\hat{E}$ is that, if we compute a Neighborhood Cover in graph $C \setminus \hat{E}$, with distance bound $\hat{D}/(1024 \log^4 W)$, using, for example, Procedure InitNC, then the weight of each resulting cluster will be significantly lower than $W(C)$. Standard balanced multicuts mentioned above provide this property as well. As already mentioned, an advantage of pseudocuts is that, as we show below, we can efficiently compute a pseudocut $\hat{E}$ in graph C, together with an expander X, whose vertex set $V(X) = \left\{t_e \mid e \in \hat{E}^*\right\}$ corresponds to a large subset $\hat{E}^* \subseteq \hat{E}$ of edges of the pseudocut, and an embedding of X into C with low congestion, such that every edge of X is embedded into a short path. Intuitively, if the size of the pseudocut $\hat{E}$ is sufficiently large, then an algorithm for the MaintainCluster problem can proceed as follows: we maintain an ES-Tree whose root is a new vertex s, that connects to every endpoint of every edge $e \in \hat{E}$ with $t_e \in V(X)$. This allows us to ensure that every vertex of C is sufficiently close to some edge $e \in \hat{E}$ with $t_e \in V(X)$, and to easily detect when this property no longer holds, so flag F_C can be raised. Additionally, we employ known algorithms for APSP in expander graphs, in order to ensure that the endpoints of all edges in $\left\{e \in \hat{E} \mid t_e \in V(X)\right\}$ remain close to each other. Lastly, as graph C undergoes update operations, we can maintain the expander X and its embedding into C using the "expander pruning" algorithm from Theorem 2.2. This algorithm ensures that, if $|\hat{E}|$ is sufficiently large, then expander X can be maintained over a long enough sequence of update operations of the input structure $\mathcal{I}$. Once expander X can no longer be maintained (that is, a long enough sequence of update operations has occurred since X was created), we recompute the pseudocut and the corresponding expander, together with its embedding, from scratch. This is precisely the approach that we employ, if the size of the pseudocut that we compute is large enough. However, if $|\hat{E}|$ is small, then, using this approach, we may need to recompute the expander and the pseudocut too often, resulting in high running time. We discuss our approach for dealing with this situation later. We now proceed to define the main tools that we use.

5.2 Expanders and Their Embeddings

Recall that a graph X is a φ-expander, for a parameter $0 < \varphi < 1$, iff for every partition (A, B) of $V(X)$, $|E_X(A, B)| \geq \varphi \min\{|A|, |B|\}$ holds.

Definition (Embeddings of Graphs) *Let G, X be two graphs with $V(X) \subseteq V(G)$. An* embedding *of X into C is a collection $\mathcal{P} = \{P(e) \mid e \in E(X)\}$ of paths in graph G, such that, for every edge $e = (x, y) \in E(X)$, path $P(e)$ connects vertex x to vertex y. The* congestion *of the embedding is the maximum, over all edges $e' \in E(G)$, of the number of paths in $\mathcal{P}$ containing e'.*

Assume now that we are given a subset $\hat{E} \subseteq E(C)$ of edges of the graph C. We let $C_{|\hat{E}}$ be the graph obtained from C by subdividing every edge $e = (u, v) \in \hat{E}$ with a vertex t_e. We set the lengths of the new edges $\ell(u, t_e) = \ell(v, t_e) = \ell(u, v)$.

Definition (Expander over an Edge Subset) *Given a subset $\hat{E} \subseteq E(C)$ of edges of the graph C, and another graph X, we say that X is defined over the set $\hat{E}$ of edges if $V(X) = \left\{t_e \mid e \in \hat{E}\right\}$.*

5.3 Computing a Pseudocut, the Corresponding Expander, and its Embedding

The following theorem is a central tool that we use to design efficient algorithms for the MaintainCluster problem.

Theorem 5.1 *There is a deterministic algorithm, that we call* AlgPseudocut&Expander*, whose input is a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, where C is a connected graph, with arbitrary weights*

$w(x) \geq 0$ for vertices $x \in V(C)$, and parameters $0 < \epsilon < 1$, $\hat{W} \geq \max\left\{W(C), |E(C)|, 2^{\Omega(1/\epsilon^2)}\right\}$, and $\hat{D} > D$, such that, for every vertex $x \in V(C)$, $w(x) \leq \frac{W(C)}{4\hat{W}^\epsilon}$ holds. The algorithm computes a $(\hat{D}, \rho)$-pseudocut $\hat{E}$ in C, with $\rho = \hat{W}^\epsilon$, and a φ^*-expander X defined over an edge set $\hat{E}^* \subseteq \hat{E}$, with $|\hat{E}^*| \geq \Omega(|\hat{E}|/\hat{W}^{2\epsilon})$, such that the maximum vertex degree in X is $O(\log \hat{W})$, and $\varphi^* = 1/(\log \hat{W})^{O(1/\epsilon)}$. The algorithm also computes an embedding $\mathcal{P}$ of X into $C_{|\hat{E}^*}$ with congestion at most $\eta = \hat{D} \cdot \hat{W}^\epsilon (\log \hat{W})^{O(1/\epsilon)}$, such that the length of every path in $\mathcal{P}$ is at most $O(\hat{D} \log^2 \hat{W})$. The running time of the algorithm is $O\left(|E(C)| \cdot \hat{D}^2 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$.

We emphasize that the expander X that we construct is unweighted. Therefore, when we talk about the length of a path P in the expander X, we refer to the number of edges on P.

Proof of Theorem 5.1. The following theorem is key to the proof of Theorem 5.1.

Theorem 5.2 *There is a deterministic algorithm, whose input consists of:*

- *a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, where C is a connected graph;*
- *arbitrary weights $w(x) \geq 0$ for vertices $x \in V(C)$;*
- *parameters $0 < \epsilon < 1$, $\hat{W} \geq \max\left\{W(C), |E(C)|, 2^{\Omega(1/\epsilon^2)}\right\}$, and $\hat{D} > D$; and*
- *a $(\hat{D}, \rho)$-pseudocut $\hat{E}$ for C of cardinality k, with $\rho = \hat{W}^\epsilon$.*

The algorithm computes one of the following:

- *either a φ^*-expander X defined over a subset $\hat{E}^* \subseteq \hat{E}$ of at least $\Omega(k/\hat{W}^{2\epsilon})$ edges, with $\varphi^* = 1/(\log \hat{W})^{O(1/\epsilon)}$, of maximum vertex degree $O(\log \hat{W})$, together with an embedding $\mathcal{P}$ of X into $C_{|\hat{E}^*}$ with congestion at most $\hat{D} \cdot \hat{W}^\epsilon (\log \hat{W})^{O(1/\epsilon)}$, such that the length of every path in $\mathcal{P}$ is at most $O(\hat{D} \log^2 \hat{W})$; or*
- *another $(\hat{D}, \rho)$-pseudocut $\hat{E}'$ in C, with $|\hat{E}'| \leq |\hat{E}| \left(1 - \frac{1}{\hat{W}^\epsilon (\log \hat{W})^{O(1/\epsilon)}}\right)$.*

The running time of the algorithm is $O\left(|E(C)| \cdot \hat{D}^2 \cdot \hat{W}^{O(\epsilon)} (\log \hat{W})^{O(1/\epsilon^2)}\right)$.

The proof of Theorem 5.1 easily follows from Theorem 5.2. Throughout, we set $\rho = \hat{W}^\epsilon$. Recall that for every vertex $x \in V(C)$, $w(x) \leq \frac{W(C)}{4\hat{W}^\epsilon} \leq \frac{W(C)}{4\rho}$ holds. We start with an initial $(\hat{D}, \rho)$-pseudocut $\hat{E}_0$, that is constructed as follows. We construct a partition $S_1, \ldots, S_r$ of $V(C)$, where $r = 4\rho$, and for all $1 \leq i \leq r$, $W(S_i) \leq W(C)/\rho$ via a simple greedy algorithm: start with $S_1 = S_2 = \cdots = S_r = \emptyset$, and then process the vertices of C in the order of their weights from highest to lowest, breaking ties arbitrarily. When vertex x is processed, we add it to the set S_i, whose current weight is the smallest (if several sets have the same weight, choose one arbitrarily). It is immediate to verify that, when the algorithm terminates, $\max_{1 \leq i < j \leq r} |W(S_i) - W(S_j)| \leq 2 \max_{x \in V(C)} w(x) \leq \frac{W(C)}{2\rho}$. Since $r = 4\rho$, there must be some set S_i with $W(S_i) \leq W(C)/(4\rho)$. We conclude that for every set S_j, $W(S_j) \leq W(C)/\rho$. We then let $\hat{E}_0$ contain all edges whose endpoints lie in different sets $S_1, \ldots, S_r$. It is immediate to verify that $\hat{E}_0$ is a $(\hat{D}, \rho)$-pseudocut (in fact it is a valid (D^*, ρ)-pseudocut for any parameter $D^* < \infty$).

We then perform iterations. The input to the ith iteration is the current $(\hat{D}, \rho)$-pseudocut $\hat{E}_{i-1}$, where the input to the first iteration is the pseudocut $\hat{E}_0$ that we have just computed. We apply the algorithm from Theorem 5.2 to the same input $\mathcal{I}$, and $(\hat{D},\rho)$-pseudocut $\hat{E}_{i-1}$. If the outcome of the algorithm is another $(\hat{D},\rho)$-pseudocut $\hat{E}'$ with $|\hat{E}'| \leq |\hat{E}_{i-1}|\left(1-\frac{1}{\hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}}\right)$, then we set $\hat{E}_i=\hat{E}'$, and continue to the next iteration. Otherwise, the algorithm must return a φ^*-expander X defined over an edge set $\hat{E}^* \subseteq \hat{E}_{i-1}$, for $\varphi^*=1/(\log \hat{W})^{O(1/\epsilon)}$, with $|\hat{E}^*| \geq \Omega(|\hat{E}_{i-1}|/\hat{W}^{2\epsilon})$, such that the maximum vertex degree in X is at most $O(\log \hat{W})$. The algorithm also computes an embedding $\mathcal{P}$ of X into C with congestion at most $\hat{D}\cdot \hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}$, such that the length of every path in $\mathcal{P}$ is at most $O(\hat{D}\log^2 \hat{W})$. We then terminate the algorithm, and return $\hat{E}_{i-1}, X$ and $\mathcal{P}$ as its outcome.

Since we are guaranteed that, for all i, $|\hat{E}_i| \leq |\hat{E}_{i-1}|\left(1-\frac{1}{\hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}}\right)$, and $|E(C)| \leq \hat{W}$, the number of iterations is bounded by $\hat{W}^{\epsilon}\cdot(\log \hat{W})^{O(1/\epsilon)}$. Since the running time of each iteration is $O\left(|E(C)|\cdot \hat{D}^2\cdot \hat{W}^{O(\epsilon)}(\log \hat{W})^{O(1/\epsilon^2)}\right)$, the total running time of the algorithm is bounded by:

$$O\left(|E(C)|\cdot \hat{D}^2\cdot \hat{W}^{O(\epsilon)}\cdot (\log \hat{W})^{O(1/\epsilon^2)}\right).$$

In order to complete the proof of Theorem 5.1, it is now enough to prove Theorem 5.2. The proof is somewhat technical, and we provide it in Section E.1 of Appendix. We provide here a high-level overview of the proof. A key part of the proof is an algorithm that either computes the desired expander X and its embedding, or it computes a collection $E_1,\ldots,E_h$ of disjoint subsets of edges from $\hat{E}$, such that the cardinality of each subset is at least $k' \geq k/(\hat{W}^{O(\epsilon)}(\log \hat{W})^{O(1/\epsilon)})$ and $h \geq \rho$. The algorithm also computes a subset E' of at most $k'/2$ edges of C, such that, in graph $C\setminus E'$, for every pair E_i,E_j of distinct subsets of edges, the length of the shortest path connecting an edge of E_i to an edge of E_j is at least $10\hat{D}$. This is sufficient for us in order to define the new pseudocut, $\hat{E}'=(\hat{E}\setminus E_i)\cup E'$, where set E_i is carefully chosen to ensure that $\hat{E}'$ is indeed a valid pseudocut. Intuitively, E_i is chosen so that the total weight of all vertices in the ball of radius $2\hat{D}$ around E_i in graph $C\setminus E'$ is the smallest.

The algorithm mentioned above, that either computes the expander X with its embedding, or the desired collection $E_1,\ldots,E_h$ of subsets of $\hat{E}$, proceeds by iteratively applying the cut-matching game of [KRV09, KKOV07] to graph $C_{|\hat{E}}$. Using standard techniques, such an algorithm can be used in order to either compute an expander X over a large enough subset $\hat{E}^* \subseteq \hat{E}$ of edges and to embed it into $C_{|\hat{E}}$, or to compute two large enough subsets $\hat{E}_1,\hat{E}_2$ of edges of $\hat{E}$, together with another relatively small subset E' of edges of C, such that, in $C\setminus E'$, every path connecting an endpoint of an edge of $\hat{E}_1$ to an endpoint of an edge of $\hat{E}_2$ is sufficiently long. We start by applying this algorithm to the initial edge set $\hat{E}$ in graph $C_{|\hat{E}}$. If the algorithm returns the desired expander and its embedding, then we are done. Otherwise, it returns two subsets $\hat{E}_1,\hat{E}_2$ of edges of $\hat{E}$, and edge set E'. We then recursively apply the same algorithm to both $\hat{E}_1$ and $\hat{E}_2$, instead of $\hat{E}$. Again, if the outcome is an expander and its embedding into $C_{|\hat{E}}$ then we are done; otherwise, we obtain sufficiently large subsets $\hat{E}_1^1,\hat{E}_1^2$ of $\hat{E}_1$, and $\hat{E}_2^1,\hat{E}_2^2$ of $\hat{E}_2$. We then continue applying the same algorithm to each of the four resulting subsets of $\hat{E}$. This process continues until we either compute an expander X and its embedding into $C_{|\hat{E}}$, or the number of subsets of $\hat{E}$ reaches $h=\Omega(\rho)$. Throughout the algorithm, we maintain a subset E' of edges of C, such that, in graph $C\setminus E'$, the distance between every pair of edge sets $\hat{E}_i,\hat{E}_j$ that we maintain is at least $\Omega(\hat{D})$. Once the number of disjoint subsets of $\hat{E}$ that we maintain reaches h, if we did not terminate the algorithm with the desired expander X and its embedding, we return the resulting subsets $\hat{E}_1,\ldots,\hat{E}_h$ of $\hat{E}$, together with the edge set E' that the algorithm maintained. The complete proof of Theorem 5.2 appears in Section E.1 of Appendix. □

5.4 APSP in Expanders

It is well known (see Observation 2.1), that, if an n-vertex graph X is a φ-expander, and maximum vertex degree in X is bounded by Δ, then for any pair x, y of vertices in X, there is a path connecting them of length at most $O(\Delta \log n/\varphi)$. We need an algorithm that supports short-path queries on an expander, as it undergoes edge deletions. In each such query, given a pair x, y of vertices of X, we need to return a path connecting x to y in X of length comparable to $O(\Delta \log n/\varphi)$. As expander X undergoes edge deletions, we will also prune some vertices out of it to ensure that it remains an expander. Such an algorithm was provided in [CS21], who used techniques similar to those in [CGL+19]. However, we need a slightly different tradeoff between the approximation factor and the running time from that in [CS21], and we summarize it in the following theorem. The proof is practically identical to that of [CS21] and is deferred to Section E.2 of Appendix.

Theorem 5.3 *(Analogue of Theorem 3.13 in [CS21]) There is a deterministic algorithm that, given an n-vertex (unweighted) graph X that is a φ-expander with maximum vertex degree at most Δ, and a parameter $0 < \epsilon < 1$, such that $\varphi \leq 1/2^{\Omega(1/\epsilon)}$, with graph X undergoing a sequence at most $\varphi|E(X)|/(20\Delta)$ edge deletions, maintains a vertex set $S \subseteq V(X)$, such that, for every $t > 0$, after t edges are deleted from X, $|S| \leq O(t\Delta/\varphi)$ holds. Vertex set S is incremental, so vertices may join it but they may not leave it. The algorithm also supports queries* expander-short-path-query*: given a pair of vertices $x, y \in V(X) \setminus S$, return an x-y path P in $X \setminus S$ of length at most $O\left(\Delta^2 (\log n)^{O(1/\epsilon^2)}/\varphi\right)$, with query time $O(|E(P)|)$. The total update time of the algorithm is $O\left(\frac{n^{1+O(\epsilon)}\Delta^7(\log n)^{O(1/\epsilon^2)}}{\varphi^5}\right)$.*

5.5 Maintaining the Expander

Suppose we are given some φ-expander X that is defined over some subset $\hat{E}^*$ of edges of C, with $|\hat{E}^*| = k$, and an embedding $\mathcal{P}$ of X into $C_{|\hat{E}^*}$, with congestion at most η. Next, we show that this expander can "withstand" a long sequence of valid update operations. In order to do so, we need to define a *cost* of each valid update operation.

Definition (Update operation costs) *The costs of valid update operations are defined as follows. The cost of an edge-deletion is* 1*, and the cost of every other valid update operation is* 0*.*

The following observation is immediate.

Observation 5.4 *Let $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$ be a valid input structure, undergoing a sequence Σ of valid update operations, with dynamic degree bound μ. Then the total cost of all update operations in Σ is at most $O(N^0(C)\mu)$, where $N^0(C)$ is the number of the regular vertices in the initial graph C.*

The observation follows from the fact that the total number of edges that are ever present in C is bounded by $N^0(C)\mu$. Lastly, we use the following theorem in order to maintain the expander X computed by Theorem 5.1.

Theorem 5.5 *There is a deterministic algorithm, that we call* AlgMaintainExpander*, that receives as input the following:*

- *a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, where C is a connected graph with maximum vertex degree at most μ;*

- *parameters $0 < \epsilon < 1$, $\hat{W} \geq \max\left\{N^0(C), \mu, |E(C)|, 2^{\Omega(1/\epsilon^2)}\right\}$, and $\hat{D} > D$, where $N^0(C)$ is the number of regular vertices in C at the beginning of the algorithm;*
- *a φ^*-expander X, for $\varphi^* = 1/(\log \hat{W})^{O(1/\epsilon)}$, defined over an edge set $\hat{E}^* \subseteq E(C)$ of cardinality k, such that the maximum vertex degree in X is at most $O(\log \hat{W})$;*
- *an embedding $\mathcal{P}$ of X into $C_{|\hat{E}^*}$ with congestion $\eta = \hat{D} \cdot \hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}$, where each path in $\mathcal{P}$ has length at most $O(\hat{D} \log^2 \hat{W})$; and*
- *an online sequence Σ of valid update operations of total cost at most $\frac{\varphi^* k}{c\eta \log \hat{W}}$ for some large enough constant c, with dynamic degree bound μ.*

The algorithm maintains a non-empty subgraph $X' \subseteq X$, and supports expander-short-path queries: given a pair of vertices $x, y \in V(X')$, return an x-y path P in graph $C_{|\hat{E}^*}$, of length at most $O\left(\hat{D} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$, with query time $O(|E(P)|)$. Graph X' is decremental, so vertices and edges may be deleted from it, but they may never be added to it. The total update time of the algorithm is bounded by:

$$O\left(N^0(C) \cdot \mu \cdot \hat{D}^2 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right).$$

Note that, by substituting the parameters φ^* and η, Algorithm AlgMaintainExpander can maintain the graph X' over the course of a sequence of valid update operations of total cost at most $\Omega\left(\frac{k}{\hat{D} \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}}\right)$.

Proof: Note that, since graph C is connected, $k \leq |E(C)| \leq N^0(C)\mu$ must hold. We use the algorithm from Theorem 5.3 in order to maintain the expander X under edge deletions, with parameters ϵ, $\varphi = \varphi^*$, and $\Delta = O(\log \hat{W})$. For every edge $e \in E(C)$, we maintain a list $L(e)$ of all edges $e' \in E(X)$ with $e \in P(e')$, where $P(e') \in \mathcal{P}$ is the path embedding e'. (If edge e lies in $\hat{E}^*$, then it is split into two edges by vertex t_e in graph $C_{|\hat{E}^*}$; we then include in $L(e)$ all edges of X whose embedding contains either one of the resulting edges.) As each such list $L(e)$ contains at most $\eta = \hat{D} \cdot \hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}$) edges, initializing all lists $L(e)$ takes time $O(|E(X)| \cdot \eta) \leq O(|E(C)| \cdot \eta^2) \leq O(N^0(C)\mu\hat{D}^2\hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)})$.

We handle update operations performed on cluster C as follows. First, if an isolated vertex is deleted from C, or a supernode splitting operation is executed, then we do nothing. Assume now that an edge e is deleted from C. Then for every edge $e' \in L(e)$, we delete e' from the graph X, and update the data structure from Theorem 5.3 accordingly. Note that, if the cost of an update operation is a, then the number of edges deleted from X as the result of this operation is at most $2\eta a$. Therefore, if we perform a sequence of valid update operations in graph C, whose total cost is at most $\varphi^* k/(c\eta \log \hat{W})$, this will result in a sequence of edge deletions to the data structure from Theorem 5.3, whose length is at most $2\varphi^* k/(c \log \hat{W}) \leq O(\varphi^*|E(X)|/(c\Delta))$, where Δ is the maximum vertex degree in X. By letting c be a large enough constant, we can ensure that this is bounded by $\varphi^*|E(X)|/(20\Delta)$. The graph X' that we maintain is defined as follows. Consider some time t in the algorithm, after t updates to graph C were executed. Let $M(t)$ denote the number of edge deletions from graph X that were executed so far, and let E_t be the corresponding set of edges. Let S_t be the set S that the algorithm from Theorem 5.3 maintains, after the edges in E_t are deleted from X. Then graph X' at time t is defined to be $(X \setminus E_t) \setminus S_t$. Observe that Theorem 5.3 ensures that graph X' non-empty, since for all t, $|S_t| \leq O(|E_t|\Delta/\varphi^*) \leq O(\varphi^* k\Delta/(c\varphi^* \log \hat{W}) \leq O(k\Delta/(c \log \hat{W}))$; letting c be a large enough constant, we can ensure that this number is below $k/2$. The total update time of the algorithm from Theorem 5.3 is bounded by: $O\left(\frac{k^{1+O(\epsilon)}\Delta^7(\log k)^{O(1/\epsilon^2)}}{(\varphi^*)^5}\right) \leq O\left((N^0(C)\mu)^{1+O(\epsilon)}(\log \hat{W})^{O(1/\epsilon^2)}\right) \leq O\left(N^0(C)\mu\hat{W}^{O(\epsilon)}(\log \hat{W})^{O(1/\epsilon^2)}\right)$.

Consider now expander-short-path query, in which we are given a pair of vertices $x, y \in V(X')$, and we need to return an x-y path P in graph $C_{|\hat{E}^*}$, of length at most $O\left(\hat{D} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$, with query time $O(|E(P)|)$. We run query expander-short-path-query(x, y) in graph X', using the algorithm from Theorem 5.3, obtaining an x-y path Q in X', of length at most $O\left(\Delta^2 (\log k)^{O(1/\epsilon^2)}/(\varphi^*)^5\right) \leq O\left((\log \hat{W})^{O(1/\epsilon^2)}\right)$, in time $O(|E(Q)|)$. Next, we use, for each edge $e \in E(Q)$, the embedding path $P(e) \in \mathcal{P}$, whose length is $O(\hat{D} \log^2 \hat{W})$, to replace the edge e on the path Q. As the result, we obtain a path Q' in graph $C_{|\hat{E}^*}$ connecting x to y, of length $O\left(\hat{D} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$. The query time of the algorithm is $O(|E(Q')|)$. □

5.6 A Faster Algorithm for the MaintainCluster Problem

The tools that we have developed so far are already sufficient in order to obtain a faster algorithm for the MaintainCluster problem. Since we do not use this algorithm in our final result, we only provide its sketch here, starting with some intuition.

Intuitively, an algorithm for the MaintainCluster problem, on a given input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$ undergoing a sequence of valid update operations with dynamic degree bound μ can proceed as follows. We set the weight of every regular vertex to be 1, and of every supernode to 0. We then apply Algorithm AlgPseudocut&Expander to compute a $(\hat{D}, \rho)$-pseudocut $\hat{E}$, for $\hat{D} = \Theta(D \log^4 \hat{W})$, and $\rho = \hat{W}^{\epsilon}$, where $\hat{W} = W\mu$, and W is the given input parameter bounding the total number of regular vertices in the initial graph C. We also compute the corresponding expander X defined over a large subset $\hat{E}^* \subseteq \hat{E}$ of edges, and its embedding $\mathcal{P}$ into $C_{|\hat{E}^*}$. We then initialize Algorithm AlgMaintainExpander on the expander X, that will maintain a subgraph $X' \subseteq X$. Additionally, we construct an ES-Tree in the graph $C_{|E^*}$, whose root vertex is a new vertex s, that connects with an edge to every vertex of $V(X')$. The depth threshold of the ES-Tree is $2^{13} D \log^4 \hat{W}$. The ES-Tree will allow us to ensure that every vertex of C remains close to one of the vertices of $V(X')$; whenever this is not the case, this data structure allows us to detect it and to provide a violating pair v, v' of regular vertices with $\mathsf{dist}_C(v, v') > 1024 D \log^4 \hat{W}$. We are also guaranteed by Theorem 5.5, that every pair of vertices in the expander X has a short path connecting them in $C_{|E^*}$. This ensures that, for every pair $e, e' \in \hat{E}^*$ of edges, there is a short path connecting their endpoints in C. These data structures can also easily support queries short-path-query for pairs of vertices in cluster C, that are required from an algorithm for the MaintainCluster problem. As observed already, this data structure can withstand a sequence of update operations to cluster C, whose cost is $\Omega\left(\frac{|\hat{E}^*|}{\hat{D} \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}}\right)$, and the total update time required to maintain the data structure is $O\left(N^0(C) \cdot \mu \cdot \hat{D}^2 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$.

Once a sequence of update operations of cost $\Theta\left(\frac{|\hat{E}^*|}{\hat{D} \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}}\right)$ is performed in cluster C, we discard the current data structures, run the algorithm AlgPseudocut&Expander again, and recompute the data structures from scratch. We call the execution of the algorithm between a pair of successive calls to AlgPseudocut&Expander a single phase. If we are lucky, and every time that Algorithm AlgPseudocut&Expander is called, we obtain a pseudocut $\hat{E}$, whose cardinality is at least $|E(C)|/\hat{W}^{\epsilon}$, then the number of phases will be bounded by $O\left(\hat{W}^{O(\epsilon)} \mu D (\log \hat{W})^{O(1/\epsilon)}\right)$. The total update time of this algorithm will then be at most

$$O\left(N^0(C) \mu^2 D^3 \hat{W}^{O(\epsilon)} (\log \hat{W})^{O(1/\epsilon^2)}\right).$$

However, we may not be so lucky, and it is possible that at some point Algorithm AlgPseudocut&Expander

may return a pseudocut $\hat{E}$ of small cardinality.

One straightforward way to deal with this issue is to initialize an ES-Tree in graph C, rooted at each regular vertex that serves as an endpoint of an edge in $\hat{E}$; the depth each tree is $2^{13}D\log^4\hat{W}$. Once the root of a tree is deleted from C, we say that the tree is destroyed. As long as not all such trees are destroyed, we can implement an algorithm for the MaintainCluster problem, similarly to AlgSlow. Once every tree is destroyed, we are guaranteed that all edges of $\hat{E}$ are deleted from C. From the properties of the pseudocut, and since we guarantee that the diameter of C remains below $2^{13}D\log^4\hat{W} < \hat{D}$, this means that the total number of all regular vertices in C has decreased by at least factor ρ. We can then restart the algorithm outlined above for the MaintainCluster problem in C from scratch. This approach provides a faster algorithm than AlgSlow, since we no longer have an ES-Tree data structure rooted at every single regular vertex of C, and since $|\hat{E}|$ is relatively small. In fact, in order to optimize the running time of the algorithm, we should use a threshold of $\sqrt{N^0(C)}$ for $|\hat{E}|$: as long as $|\hat{E}| \geq \sqrt{N^0(C)}$, we use the expander-based approach, and once $|\hat{E}|$ falls below this value, we use the ES-Tree's; this will result in running time that is roughly $(N^0(C)\mu)^{1.5+O(\epsilon)}D^3(\log(W\mu))^{O(1/\epsilon)}$ for the MaintainCluster problem. One advantage of this approach is that it avoids recursion. We roughly estimate that, using this algorithm instead of Theorem 4.14 would result in an algorithm for APSP with total update time approximately $O\left(m^{1.5+O(\epsilon)}(\log m)^{O(1/\epsilon^2)}\right)$, and approximation factor $(\log m)^{O(1/\epsilon)}$. This already gives a somewhat fast algorithm for sparse graphs, but this running time is still much higher than our desired bound.

6 Third Main Tool: Good Clusters

Throughout this section, we assume that we are given an input to the MaintainCluster problem, that consists of a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e\in E(C)}, D\right)$, where C is a connected graph undergoing a sequence Σ of valid update operations with a dynamic degree bound μ. We refer to D as *target distance threshold for C*. We are also given a parameter $\hat{W} \geq N^0(C)\cdot\mu$, where $N^0(C)$ is the number of regular vertices of C at the beginning of the algorithm, and a precision parameter $0 < \epsilon < 1$. We also use a parameter $\hat{D} = 2^{30}D\log^{10}\hat{W}$.

Throughout this section, we assume that $D \leq \hat{W}$, and $\hat{W} \geq 2^{\Omega(1/\epsilon^2)}$, since otherwise, from Observation 4.16, algorithm AlgSlow provides the desired bounds on the running time, with approximation factor $O(\log^4\hat{W})$. We define vertex weights as usual: the weight of each regular vertex is 1, and the weight of each supernode is 0. We denote by $W(C)$ the total weight of all vertices in C. We are then guaranteed that, throughout the execution of the update sequence Σ, $\hat{W} \geq \max\left\{W(C)\mu, |E(C)|, 2^{\Omega(1/\epsilon^2)}\right\}$ holds.

Throughout the remainder of this section, we also use the following parameters from Theorem 5.1: $\rho = \hat{W}^\epsilon$, $\varphi^* = 1/(\log\hat{W})^{O(1/\epsilon)}$, and $\eta = \hat{D}\cdot\hat{W}^\epsilon(\log\hat{W})^{O(1/\epsilon)}$. For convenience, all these parameters are summarized in Section 9.

In this section, we will define a new problem, called MaintainGoodCluster, which is an easier version of the MaintainCluster problem. In this new problem, we will only need to maintain the cluster C and support the short-path-query(C, v, v') queries as long as the cluster is "good" (we define what it means below). Once the cluster is no longer good, the algorithm can raise a flag F^b_C to indicate that, and then the algorithm terminates. However, when flag F^b_C is raised, the algorithm is required to provide a "bad witness" for the fact that C is no longer a good cluster, which is a small $(\hat{D}, \rho)$-pseudocut of C.

We start by defining good clusters and bad witnesses in Section 6.1. We then define the MaintainGoodCluster problem and provide an algorithm for it in Section 6.2. All main parameters that are used in this section are summarized in Section 9.

6.1 Good Clusters and Witnesses

Assume that we are given a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, parameters μ, $0 < \epsilon < 1$, and $\hat{W} \geq N^0(C) \cdot \mu$, that we use in order to define parameters $\hat{D}$, ρ, φ^* and η, as described above (see also Section 9).

Definition (Good Clusters and Witnesses) *We say that C is a* type-1 good cluster *iff $W(C) \leq \hat{W}^{10\epsilon}$. We say that C is a* type-2 good cluster *iff there is a collection $\hat{E}^* \subseteq E(C)$ of edges of C of cardinality $k \geq \Omega(W(C)/\hat{W}^{3\epsilon})$, and a φ^*-expander X defined over the edges of $\hat{E}^*$, with maximum vertex degree at most $O(\log \hat{W})$, together with an embedding $\mathcal{P}$ of X into $C_{|\hat{E}^*}$ with congestion at most η, such that the length of every path in $\mathcal{P}$ is at most $O(\hat{D} \log^2 \hat{W})$. We call $(\hat{E}^*, X, \mathcal{P})$* a good witness *for C.*

We will informally refer to a cluster that is type-1 or type-2 good as a good cluster. Note that a cluster may be a type-2 good cluster, but this does not guarantee that we can efficiently find a good witness for it; generally when we establish that a cluster is a type-2 good cluster, we will also provide a good witness for it.

Definition (Bad Witness) *At any point during the execution of the valid update sequence Σ on C, we say that a set $\hat{E}$ of edges is a* bad witness *for C iff $|\hat{E}| < W(C)/\hat{W}^{\epsilon}$, and $\hat{E}$ is a $(\hat{D}, \rho)$-pseudocut for C, where, as we defined before, $\hat{D} = 2^{30} D \log^{10} \hat{W}$, and $\rho = \hat{W}^{\epsilon}$.*

(Notice that it is possible that a cluster is a good cluster and yet it has a bad witness). We need the following observation that easily follows from Theorem 5.1.

Observation 6.1 *There is a deterministic algorithm, that, given a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, where C is a connected graph, with weights $w(v) = 1$ on its regular vertices and $w(u) = 0$ on supernodes, together with parameters $0 < \epsilon < 1$ and $\hat{W} \geq \max\left\{W(C), |E(C)|, 2^{\Omega(1/\epsilon^2)}\right\}$, either (i) correctly establishes that C is a type-1 good cluster; or (ii) correctly establishes that C is a type-2 good cluster and produces a good witness for it; or (iii) computes a bad witness $\hat{E}$ for C. The running time of the algorithm is $O\left(|E(C)| \cdot \hat{D}^2 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$.*

Proof: Checking whether C is a type-1 good cluster can be easily done in time $O(|E(C)|)$. Assume now that C is not a type-1 good cluster. Since the weight of every regular vertex is 1, and $W(C) \geq \hat{W}^{10\epsilon}$, we get that for every vertex $x \in V(C)$, $w(x) \leq \frac{W(C)}{4\hat{W}^{\epsilon}}$. We apply Algorithm AlgPseudocut&Expander from Theorem 5.1 to the cluster C. Consider the $(\hat{D}, \rho)$-pseudo-cut $\hat{E}$ that that the algorithm computes. If $|\hat{E}| < W(C)/\hat{W}^{\epsilon}$, then $\hat{E}$ is a bad witness for C. Otherwise, the algorithm computes an edge set $\hat{E}^* \subseteq \hat{E}$, with $|\hat{E}^*| \geq \Omega(|\hat{E}|/\hat{W}^{2\epsilon}) \geq \Omega(W(C)/\hat{W}^{3\epsilon})$, and a φ^*-expander X defined over the edge set $\hat{E}^*$, such that the maximum vertex degree in X is $O(\log \hat{W})$. The algorithm also computes an embedding $\mathcal{P}$ of X into $C_{|\hat{E}^*}$ with congestion at most η, such that the length of every path in $\mathcal{P}$ is at most $O(\hat{D} \log^2 \hat{W})$. Clearly, $(\hat{E}^*, X, \mathcal{P})$ is a good witness for C. Lastly, the running time of AlgPseudocut&Expander is $O\left(|E(C)| \cdot \hat{D}^2 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$. □

6.2 MaintainGoodCluster Problem and an Algorithm for It

In this subsection we define the MaintainGoodCluster problem, that is identical to the MaintainCluster problem, except that now the algorithm may raise flag F^b_C at any point, to indicate that cluster C is not a good cluster. When raising flag F^b_C, the algorithm must also supply a bad witness for C. The algorithm then terminates.

Definition (MaintainGoodCluster Problem) *In the* MaintainGoodCluster *problem, we are given a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e\in E(C)}, D\right)$, together with weights $w(v) = 1$ on regular vertices $v \in V(C)$, and weights $w(u) = 0$ on supernodes $u \in V(C)$, and parameters $\mu \geq 1$, $0 < \epsilon < 1$, and $\hat{W} \geq \max\left\{N^0(C)\mu, 2^{\Omega(1/\epsilon^2)}\right\}$. Graph C undergoes a sequence Σ of valid update operations with dynamic degree bound μ. The goal of the algorithm is to support queries* short-path-query(C, v, v')*: given a pair $v, v' \in V(C)$ of regular vertices of C, return a path P of length at most $O\left(D \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$ connecting them in C, in time $O(|E(P)|)$. Additionally, the algorithm may, at any time, raise a flag F_C. When the algorithm raises flag F_C, it must supply a pair $\hat{v}, \hat{v}'$ of regular vertices of C, with* $\mathsf{dist}_C(\hat{v}, \hat{v}') > 1024 D \log^4 \hat{W}$*. Once flag F_C is raised, the algorithm will obtain, as part of its input update sequence, a sequence Σ' of update operations called flag-lowering sequence, at the end of which either $\hat{v}$ or $\hat{v}'$ are deleted from C. Flag F_C is lowered after these updates are processed by the algorithm. Lastly, the algorithm may raise, at any time, flag F^b_C. When flag F^b_C is raised, the algorithm needs to supply a bad witness $\hat{E}$ for C. Once the flag F^b_C is raised, the algorithm terminates. Queries* short-path-query *may only be asked when flags F_C, F^b_C are down.*

In the following theorem we provide an efficient algorithm for the MaintainGoodCluster problem.

Theorem 6.2 *There is a deterministic algorithm, that we call* AlgMaintainGoodCluster *for the* MaintainGoodCluster *problem, whose total update time, on input cluster C is:*

$$O\left(N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right).$$

Proof: We define the parameters $\hat{D}$, ρ, η, φ^* exactly as before (see also Section 9). We partition the algorithm's execution into phases. We denote by $W_j(C)$ the weight of all vertices of C at the beginning of the jth phase. At the beginning of the jth phase, we run the algorithm from Observation 6.1 in order to check whether C is a good cluster. Recall that the running time of this algorithm is:

$$O\left(|E(C)| \cdot \hat{D}^2 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right) \leq O\left(N^0(C) \cdot \mu \cdot D^2 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right).$$

We then consider three cases.

Case 1. The first case happens if the algorithm computes a bad witness $\hat{E}$ for C. In this case, we say that the phase is a bad phase. We then raise the flag F^b_C, with the bad witness $\hat{E}$, and the algorithm terminates.

Case 2. The second case happens if the algorithm establishes that C is a type-1 good cluster, that is, $W_j(C) \leq \hat{W}^{10\epsilon}$. In this case, the current phase becomes the last phase of the algorithm. From now on, we run Algorithm AlgSlow from Theorem 4.15 on cluster C. Recall that the algorithm solves the MaintainCluster problem for C, and, when flag F_C is not raised, it supports queries short-path-query(C, v, v') queries, where, given a pair $v, v' \in V(C)$ of regular vertices of C, it returns a path P of length at most $1024D \log^4 \hat{W}$ connecting them in C, in time $O(|E(P)|)$. When flag F_C is raised, the algorithm produces a pair $\hat{v}, \hat{v}'$ of regular vertices of C, with $\mathsf{dist}_C(\hat{v}, \hat{v}') > 1024D \log^4 \hat{W}$ The total update time of this algorithm is:

$$O(W_j(C))^2 \mu D \operatorname{poly}\log \hat{W}) \leq O(N^0(C) \cdot \mu \cdot D \cdot \hat{W}^{O(\epsilon)} \cdot \operatorname{poly}\log \hat{W}).$$

If Case 2 happens, then we say that the phase is type-1 good. From the above discussion, at most one type-1 good phase may happen over the course of the algorithm.

Case 3. If Case 3 happens, then the algorithm from Observation 6.1 establishes that C is a type-2 good cluster, and produces a good witness $(\hat{E}^*, X, \mathcal{P})$ for C. Recall that $\hat{E}^*$ is a set of edges of C, whose cardinality is denoted by k, and we are guaranteed that $k \geq \Omega(W_j(C)/\hat{W}^{3\epsilon})$. Additionally, X is a φ^*-expander, defined over the edges of $\hat{E}^*$, with maximum vertex degree at most $O(\log \hat{W})$. Lastly, $\mathcal{P}$ is an embedding of X into $C_{|\hat{E}^*}$ with congestion at most η, such that the length of every path in $\mathcal{P}$ is at most $O(\hat{D} \log^2 \hat{W})$.

If Case 3 happens, then we say that the current phase is a type-2 good phase. In this case, the phase continues until the total cost of update operations in the input sequence Σ performed over the course of the phase reaches $\Theta\left(\frac{\varphi^* k}{\eta \log \hat{W}}\right) = \Theta\left(\frac{k}{\hat{D} \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}}\right) = \Theta\left(\frac{k}{D \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}}\right)$. Since $k \geq \Omega(W_j(C)/\hat{W}^{3\epsilon})$, the total cost of update operations over the course of a type-2 good phase is then at least $\Omega\left(\frac{W_j(C)}{D \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}}\right)$.

We need the following simple observation:

Observation 6.3 *The total number of type-2 good phases over the course of the algorithm is at most* $O\left(\mu \cdot D \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}\right)$.

Proof: Recall that a single type-2 good phase is executed as long as the total cost of update operations performed over the course of the phase does not exceed $\Omega\left(\frac{W(C)}{D \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}}\right)$, where $W(C)$ is the weight of all vertices of C at the beginning of the phase. Recall also that, from Observation 5.4, the total cost of all update operations performed on the cluster C over the course of the algorithm is at most $O(W^0(C)\mu)$.

Consider some integer i, and let t_i be the first time during the algorithm's execution, when $W(C)$ fell below 2^i. Then the total cost of all operations performed on the cluster C from time t_i onward is at most $O(2^i \mu)$ (this is because we can consider the cluster C at time t as a "fresh" cluster that undergoes the same sequence of update operations, and apply Observation 5.4 to that cluster). Let t_{i+1} be the first time when $W(C)$ falls below 2^{i+1}, and let T be the time period between t_i and t_{i+1}. Let $W' \leq 2^i$ be the total weight of all vertices of C at time t_i. Then the total number of type-2 good phases during the time period T is bounded by:

$$O\left(\frac{W' \mu}{W'/(D \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)})}\right) \leq O\left(\mu \cdot D \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}\right).$$

Since we only need to consider integers i between $\log W^0(C) \leq \log \hat{W}$ and 1, we get that the total number of type-2 good phases over the course of the algorithm is at most $O\left(\mu \cdot D \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}\right)$.
□

The algorithm for the type-2 good phase proceeds as follows. First, we use Algorithm AlgMaintainExpander from Theorem 5.5, on C, in order to maintain a non-empty subgraph $X' \subseteq X$ over the course of the phase. Recall that the update time of the algorithm is $O\left(W_j(C)\cdot\mu\cdot\hat{D}^2\cdot\hat{W}^{O(\epsilon)}\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right) \leq O\left(N^0(C)\cdot\mu\cdot D^2\cdot\hat{W}^{O(\epsilon)}\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right)$, and the algorithm supports expander-short-path queries: given a pair of vertices $x, y \in V(X')$, return an x-y path P in graph $C_{|\hat{E}^*}$, of length at most $O\left(\hat{D}\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right) \leq O\left(D\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right)$, with query time $O(|E(P)|)$.

Additionally, we maintain a graph C', that is obtained from the graph $C_{|\hat{E}^*}$, by adding a source vertex s, that connects, with an edge of length 1, to every vertex t_e lying in the current expander X'. We initialize a generalized ES-Tree data structure τ from Theorem 3.2 in graph C', whose root is vertex s, with depth threshold $\hat{D}+1$. We will also maintain a list L of all regular vertices of C that do not lie in the tree τ. Lastly, for every vertex $x \in V(C)\setminus L$, we maintain a pointer from x to its location in the tree τ. As before, it is immediate to generalize the algorithm from Theorem 3.2 to maintain these additional data structures. The total update time of the ES-Tree data structure is at most $O\left(W_j(C)\cdot\mu\cdot D\cdot\operatorname{poly}\log\hat{W}\right) \leq O\left(N^0(C)\cdot\mu\cdot D\cdot\operatorname{poly}\log\hat{W}\right)$.

Whenever the list L becomes non-empty, we raise the flag F_C, with the corresponding pair of vertices being v, v', where v' is any vertex in list L, and v is any regular vertex that serves as an endpoint of an edge $e \in \hat{E}$ with $t_e \in V(X')$. Notice that, since $v' \in L$, we are guaranteed that $\mathsf{dist}_{C_{|\hat{E}^*}}(v', t_e) \geq \hat{D}$. Since the length of the edge e is at most D, while $\hat{D} = 2^{30}D\log^{10}\hat{W}$, we are guaranteed that $\mathsf{dist}_C(v, v') \geq 1024D\log^4\hat{W}$, as required.

Recall that a query short-path-query(a, b) may only be asked when flag F_C is not raised, so $L = \emptyset$. Given such a query, where $a, b \in V(C)$ are regular vertices of C, we use the tree τ to compute two paths: path P' connecting a to some vertex $x' \in X'$, and path P'' connecting b to some vertex $x'' \in X'$; recall that the lengths of both paths are at most $\hat{D} = 2^{30}D\log^{10}\hat{W}$, and they can be computed in time $O(|E(P')|+O(|E(P'')|)$. Next, we run query expander-short-path-query between x' and x'' in the data structure maintained by AlgMaintainExpander, to compute a path Q in graph $C_{|\hat{E}^*}$, that connects x' to x'', of length at most $O\left(D\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right)$, with query time $O(|E(Q)|)$. The final path P connecting a to b in C is obtained by concatenating P', Q and P'', and suppressing the vertices of $\left\{t_e \mid e \in \hat{E}^*\right\}$ as needed. We are then guaranteed that the length of P is at most $O\left(D\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right)$, and, from the above discussion, query time is $O(|E(P)|)$.

Altogether, the total update time of the algorithm over the course of a single type-2 good phase is at most $O\left(N^0(C)\cdot\mu\cdot D^2\cdot\hat{W}^{O(\epsilon)}\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right)$. Since, from Observation 6.3, there are at most $O\left(\mu\cdot D\cdot\hat{W}^{O(\epsilon)}\cdot(\log\hat{W})^{O(1/\epsilon)}\right)$ type-2 good phases in the algorithm, the total update time spent on all type-2 good phases is bounded by:

$$O\left(N^0(C)\cdot\mu^2\cdot D^3\cdot\hat{W}^{O(\epsilon)}\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right).$$

Final Running Time Analysis. To summarize, the algorithm may have at most one type-1 good phase, and its running time over the course of such a phase is at most $O(N^0(C)\cdot\mu\cdot D\cdot\hat{W}^{O(\epsilon)}\cdot\operatorname{poly}\log\hat{W})$. The total update time spent on all type-2 good phases is bounded by:

$$O\left(N^0(C)\cdot\mu^2\cdot D^3\cdot\hat{W}^{O(\epsilon)}\cdot(\log\hat{W})^{O(1/\epsilon^2)}\right),$$

and the number of type-2 good phases is at most $O\left(\mu \cdot D \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon)}\right)$. Additionally, there is at most one bad phase over the course of the algorithm, and, at the beginning of each phase, the algorithm spends at most $O\left(N^0(C) \cdot \mu \cdot D^2 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right)$ in order to run the algorithm from Observation 6.1.

Therefore, the total running time of the algorithm is bounded by:

$$O\left(N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right).$$

□

7 Fourth Main Tool: Contracted Graph

In this section we develop the fourth and the last main tool that we use in our final algorithm for the MaintainCluster problem – contracted graph. Throughout this section, we assume that we are given an input to the MaintainCluster problem, that consists of a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, where C is a connected graph. Graph C undergoes a sequence Σ of valid update operations with a dynamic degree bound μ. Recall that we are guaranteed that for every regular vertex $v \in V(C)$, the total number of edges incident to v that are ever present in C is bounded by μ. Additionally, we are given a precision parameter ϵ, and a parameter $\hat{W} \geq N^0(C) \cdot \mu$, where $N^0(C)$ is the number of regular vertices of C at the beginning of the algorithm. Throughout this section, we assume that $\hat{W} \geq 2^{\Omega(1/\epsilon^2)}$, so $\hat{W} \geq \max\left\{N^0(C)\mu, 2^{\Omega(1/\epsilon^2)}\right\}$ holds. Recall that our goal is to support queries short-path-query(C, v, v'): given a pair $v, v' \in V(C)$ of regular vertices, return a path P of length at most αD, connecting them in C, in time $O(|E(P)|)$, where α is the approximation factor that the algorithm achieves. The algorithm may, at any time, raise a flag F_C, at which time it must supply a pair $\hat{v}, \hat{v}'$ of regular vertices of C, with $\mathrm{dist}_C(\hat{v}, \hat{v}') > 1024 D \log^4 \hat{W}$. Once flag F_C is raised, the algorithm will obtain, as part of its input update sequence, a sequence Σ' of update operations called flag-lowering sequence, at the end of which either $\hat{v}$ or $\hat{v}'$ are deleted from C. We use parameters $\hat{D} = 2^{30} D \log^{10} \hat{W}$, $\rho = \hat{W}^{\epsilon}$ from Section 6, and we define a new parameter $\lambda = \left\lceil \log(2048 D \log^4 \hat{W}) \right\rceil$.

We start with some intuition. Our algorithm for the MaintainCluster problem starts by running Algorithm AlgMaintainGoodCluster from Theorem 6.2 on the input $\mathcal{I}$, with the same parameters $\hat{W}$ and ϵ. As long as the algorithm does not raise the flag F^b_C, it can support queries short-path-query(C, v, v'), and it provides all required guarantees for the MaintainCluster problem, with approximation factor $(\log \hat{W})^{O(1/\epsilon^2)}$. Once the algorithm raises the flag F^b_C, it terminates, and we need to continue maintaining the cluster C by other means. Recall that, when the flag F^b_C is raised, the algorithm produces a bad witness $\hat{E}$ for C, where $\hat{E}$ is a $(\hat{D}, \rho)$-pseudocut of cardinality at most $N^0(C)/\hat{W}^{\epsilon}$. We denote by S the set of all regular vertices that serve as endpoints of the edges in $\hat{E}$; clearly, $|S| \leq N^0(C)/\hat{W}^{\epsilon}$. At this point, we can initialize an ES-Tree τ in the graph that is obtained from C, by adding a source vertex s to it, that connects to every vertex of S with a length-1 edge, and has depth $\Theta(\hat{D})$. This tree will allow us to ensure that, as long as some edge of $\hat{E}$ remains in C, every vertex of C is sufficiently close to at least one vertex in S. Once all edges of $\hat{E}$ are deleted from C, we will start the whole algorithm from scratch. When the algorithm starts from scratch, we will ensure that the diameter of C is below $\hat{D}$, by repeatedly raising the flag F^C as needed. From the definition of a pseudocut, we are then guaranteed that the total number of regular vertices in C has decreased by at least factor ρ, so we can afford this in terms of the running time. As long as $\hat{E} \neq \emptyset$, the ES-Tree τ ensures that every vertex of C is close to some vertex of S. But we also need an additional data structure to ensure that vertices of S remain close to each other. The goal of this section is to design such a data structure.

We now provide an intuitive description of the data structure. Consider the graph $\tilde{H} = C \setminus \hat{E}$. Assume that, for all $1 \leq i \leq \lambda$, we maintain a strong $(2^i, \alpha \cdot 2^i)$-neighborhood cover $\mathcal{C}_i$ for graph $\tilde{H}$, for some approximation factor $\alpha \geq 2048 \log^4 \hat{W}$. We will exploit the fact that $\hat{E}$ is a $(\hat{D}, \rho)$-pseudocut in order to ensure that, for all $1 \leq i \leq \lambda$, for every cluster $C' \in \mathcal{C}_i$, $N(C') \leq N^0(C)/\rho$ always holds (here, $N(C')$ is the number of regular vertices in C'). Additionally, we will maintain a *contracted graph* $\hat{H}$. The vertex set $\hat{H}$ consists of a set S of regular vertices – the regular vertices that serve as endpoints of the edges in $\hat{E}$, and of a set $\{u(C') \mid C' \in \bigcup_i \mathcal{C}_i\}$ of supernodes, representing the clusters in the neighborhood covers $\mathcal{C}_1, \ldots, \mathcal{C}_\lambda$. For every regular vertex $s \in S$ and super-node $u(C')$, we add an edge $e = (s, u(C'))$ to the graph if either $s \in V(C')$, or there is an edge $e' = (s, u') \in \hat{E}$, with $u' \in C'$. The length of the edge e is $2^i + \ell(e')$, where i is the index for which $C' \in \mathcal{C}_i$. One can show that, in the contracted graph $\hat{H}$, all distances between the vertices of S are approximately equal to those in the original cluster C. Moreover, since $|S| \leq N^0(C)/\hat{W}^\epsilon$, the graph is, in a sense, significantly "smaller" than C. We exploit this fact in order to solve the MaintainCluster problem recursively in graph $\hat{H}$. This will allow us to ensure that the vertices of S remain close to each other in C, and, once this is no longer the case, to raise the flag F_C, and to provide a pair of violating vertices as required. As the neighborhood covers $\mathcal{C}_1, \ldots, \mathcal{C}_\lambda$ evolve, the resulting changes to their clusters will lead to changes to the contracted graph $\hat{H}$. However, all such changes can be implemented via valid update operations of graph $\hat{H}$.

This leaves open the question of how to maintain the neighborhood covers $\mathcal{C}_i$; equivalently, we need to solve the RecDynNC problem in graph $\tilde{H} = C \setminus \hat{E}$, for every distance threshold in $\left\{2^i \mid 1 \leq i \leq \lambda\right\}$. As we have shown already, in order to obtain an algorithm for the RecDynNC problem, it is sufficient to obtain an algoritm for the MaintainCluster problem. Since all clusters in the neighborhood covers $\mathcal{C}_i$ for which we will need to solve the MaintainCluster problem will have a significantly smaller number of regular vertices than $N^0(C)$, we can maintain the neighborhood covers $\mathcal{C}_i$ by applying the algorithm for the MaintainCluster problem recursively to these much smaller clusters, that arise over the course of the algorithm for the RecDynNC problem.

This is precisely the approach that we use, except for a small caveat. Since we solve the MaintainCluster problem in the contracted graph $\hat{H}$ recursively, we need to ensure that it has a small dynamic degree bound. Consider some edge $e = (u, v) \in \hat{E}$, where v is a regular vertex, and u is a supernode. In the construction of graph $\hat{H}$ that we have described above, for every cluster $C \in \bigcup_i \mathcal{C}_i$ that contains u, we will add the edge $(v, u(C))$ to $\hat{H}$. The analysis from Section 4.4 only guarantees that every **regular** vertex of $\tilde{H}$ belongs to a small number of clusters in $\bigcup_i \mathcal{C}_i$, but we cannot guarantee it for supernodes (intuitively, this is because each supernode has weight 0; as supernodes may be added to the graph via supernode splitting, letting the weight of a supernode be non-zero would result in total weight of vertices in H increasing as well as decreasing, which would make the analysis much more difficult). We overcome this difficulty by defining the graph $\tilde{H}$ slightly differently: we start, as before, with $\tilde{H} = C \setminus \hat{E}$. But then we insert, for every edge $e = (u, v) \in \hat{E}$, where v is a regular vertex, a new *fake regular vertex* $v^F(e)$, and a new *fake edge* $e^F = (u, v^F(e))$. We view vertex $v^F(e)$ as a *fake copy* of the vertex v, and we view e^F as a fake copy of the edge e. Notice that it is possible that, for a regular vertex $v \in S$, we have introduced several fake copies – one copy for every edge of $\hat{E}$ incident to v. In the contracted graph $\hat{H}$, we connect a vertex $v \in S$ to a supernode $u(C')$ whenever cluster $C' \in \bigcup_i \mathcal{C}_i$ contains either v or any of its fake copies. Since we are guaranteed that the total number of edges incident to v in graph C always remains below μ, the number of fake copies of v in $\tilde{H}$ is also bounded by μ. As the algorithm from Section 4.4 ensures that every regular vertex of $\tilde{H}$ belongs to a small number of clusters in each set $\mathcal{C}_i$ for $1 \leq i \leq \lambda$, the dynamic degree bound for graph $\hat{H}$ will also be small (but somewhat higher than μ).

The remainder of this section is organized as follows. We define the modified graph $\tilde{H}$ in Section 7.1. We then define the corresponding contracted graph in Section 7.2. Lastly, we prove that the distances are approximately preserved in the contracted graph in Section 7.3.

7.1 Modified Graph $\tilde{H}$

Throughout, we assume that we are given a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e\in E(C)}, D\right)$, where C is a connected graph, that undergoes an online sequence Σ of valid update operations with dynamic degree bound μ. We also assume that we are given parameters $0 < \epsilon < 1$ and $\hat{W} \geq N^0(C)\mu$ as before. We use the pareameters $\hat{D} = 2^{30} D \log^{10} \hat{W}$, $\lambda = \left\lceil \log(2048 D \log^4 \hat{W}) \right\rceil$, and $\rho = \hat{W}^{\epsilon}$, that are defined exactly as before.

Lastly, we assume that, at the beginning of the algorithm, we are given a bad witness $\hat{E}$ for cluster C, so $\hat{E}$ is a $(\hat{D}, \beta)$-pseudocut, with $|\hat{E}| < N^0(C)/\hat{W}^{\epsilon}$. We denote this initial edge set $\hat{E}$ by $\hat{E}^0$, and we let S be the set of all regular vertices that serve as endpoints of edges in $\hat{E}^0$, so $|S| < N^0(C)/\hat{W}^{\epsilon}$. As cluster C undergoes a sequence of valid update operations, edge set $\hat{E}$ evolves as follows. First, if an edge $e \in \hat{E}$ is deleted from C as part of the input update operation sequence, we delete e from $\hat{E}$. Additionally, if a supernode splitting operation is performed on a supernode $u \in V(C)$, with the corresponding edge set E', then for every edge $e = (u, v) \in \hat{E} \cap E'$, we add the new copy $e' = (u', v)$ of the edge e that was created by the supernode splitting operation to set $\hat{E}$. Therefore, throughout the algorithm, an edge may leave $\hat{E}$ only if it is deleted from C, and an edge may join $\hat{E}$ only if it is a newly created copy of an edge that already lies in $\hat{E}$, where the copy is created via the supernode splitting operation. Set S of vertices always contains all regular vertices that serve as endpoints of the edges in the current set $\hat{E}$. It is immediate to verify that vertices may leave S, but they may never join S.

We now define the modified graph $\tilde{H}$, which is a dynamic graph. Initially, before any updates are applied to graph C, we define the initial graph $\tilde{H}^0$ as follows. We start with $\tilde{H}^0 = C \setminus \hat{E}^0$. Next, we process every edge $e = (v, u) \in \hat{E}^0$, where $v \in S$ is a regular vertex, one-by-one. When edge $e = (v, u)$ is processed, we insert a new regular vertex $v^F(e)$ into $\tilde{H}^0$, that we call a *fake regular vertex*, and a *fake copy of vertex* v. Additionally, we insert a new edge $e^F = (v^F(e), u)$ into $\tilde{H}^0$, whose length is $\ell(e)$. We say that $v^F(e)$ is a *fake edge*, and that it is a *fake replacement edge* for edge e. This completes the definition of the initial modified graph $\tilde{H}^0$. Observe that, for every edge $e \in \hat{E}$, there is a unique fake replacement edge for e, and for every regular vertex $s \in S$, we may have created a number of fake copies of s. In the remainder of the algorithm, we never create any new fake vertices, but we may create new fake edges, as new edges join set $\hat{E}$. We will ensure that the following invariant holds throughout the algorithm:

I1. For every edge $e = (u, v) \in \hat{E}$, there is a unique fake replacement edge e^F in $\tilde{H}$, whose one endpoint is u and the other endpoint is a fake copy of v; edge e itself does not lie in $\tilde{H}$.

For each vertex $s \in S$, let $R(s)$ be the set of all fake copies of vertex s. Since no new fake vertices are ever created after the initialization step, vertices may leave set $R(s)$, but they may never join it. Since the degree of every vertex in C is at most μ, $|R(s)| \leq \mu$ must hold for every vertex $s \in S$.

Next, we describe the updates to the graph $\tilde{H}$, as cluster C undergoes the sequence Σ of update operations. We denote by $\tilde{H}^t$ the graph $\tilde{H}$ just before the $(t+1)$th update operation $\sigma_{t+1} \in \Sigma$ is executed in graph C. The updated graph $\tilde{H}^{t+1}$ is defined as follows. If σ_{t+1} is the deletion of an isolated vertex x, then we also delete x from $\tilde{H}^t$ (where it must be an isolated vertex as well). If σ_{t+1} is the deletion of an edge e, then we proceed as follows. First, if $e \notin \hat{E}$, then $e \in E(\tilde{H})$ must hold. We delete e from $\tilde{H}^t$ as well. If $e \in \hat{E}$, then we let e^F be the unique fake replacement edge of e. We delete e^F from $\hat{E}$ and from $\tilde{H}^t$. If the endpoint v^F of e^F that is a fake regular vertex becomes an isolated vertex in $\tilde{H}$ as the result of this edge deletion, then we delete v^F from $\tilde{H}^t$ as well. If the endpoint $v \in S$ of e that is a regular vertex has no other edges of $\hat{E}$ that are incident to it, we delete v from S. Note that we have defined a sequence of valid update operations in graph $\tilde{H}$ corresponding to the update σ_{t+1}, whose length is $O(1)$ (in addition to possible updates to edge set $\hat{E}$ and vertex set S).

Lastly, assume that σ_{t+1} is a supernode-splitting operation, for some supernode u, with edge set E'. In this case, we define a new edge set E'', as follows. We process each edge $e \in E'$ one-by-one. If $e \notin \hat{E}$, then we add e to E''. Otherwise, we add to E'' the unique fake replacement edge e^F of e. We then perform the supernode-splitting operation in graph $\tilde{H}^t$, with supernode u and edge set E''. We also update edge set $\hat{E}$, as described above. Consider an edge $e = (u, v) \in E' \cap \hat{E}$. Recall that, following the supernode splitting operation in graph C, we have created a copy $e' = (u', v)$ of e in graph C, and edge e' was added to the set $\hat{E}$. Let $e^F = (u, v^F)$ be the fake replacement edge of edge e in $\tilde{H}$. Then the supernode splitting operation of u in $\tilde{H}$ has created a new edge $(e')^F = (u', v^F)$, corresponding to the edge e^F. We view edge $(e')^F$ as the fake replacement edge of the edge e'. This completes the definition of the graph $\tilde{H}^{t+1}$. Notice that, if Invariant I1 held for $\tilde{H}^t$, then it also holds for $\tilde{H}^{t+1}$. The following claim is now immediate.

Claim 7.1 *There is a deterministic algorithm that, given an online valid update sequence $\Sigma = (\sigma_1, \sigma_2, \ldots)$ for C produces, at each time $t > 0$, a sequence $\tilde{\Sigma}_t$ of valid update operations for graph $\tilde{H}$, such that $\tilde{\Sigma}_t$ contains a constant number of valid update operations, and the length of the description of each operation, as well as the time required to compute $\tilde{\Sigma}_t$, is asymptotically bounded by the length of the description of σ_t. Moreover, for all $t \geq 0$, the graph that is obtained from $\tilde{H}^0$ by applying the sequence $\tilde{\Sigma}_1 \circ \tilde{\Sigma}_2 \circ \cdots \circ \tilde{\Sigma}^t$ to it is precisely $\tilde{H}^{t+1}$. Additionally, if the dynamic graph C has dynamic degree bound μ, then the dynamic graph $\tilde{H}$ has dynamic degree bound μ as well. The algorithm also maintains the dynamic set $\hat{E}$ of edges, and the dynamic set S of vertices.*

.

7.2 Contracted Graph $\hat{H}$

A contracted graph $\hat{H}$ is a dynamic graph that is associated with the modified graph $\tilde{H}$, and the set $\hat{E}$ of edges of C. We consider some time point $t \geq 0$, and the corresponding modified graph $\tilde{H} = \tilde{H}^t$, and edge set $\hat{E} = \hat{E}^t$. We assume that we are given, for all $1 \leq i \leq \lambda$, a collection $\mathcal{C}_i$ of subgraphs of $\tilde{H}$, such that $\mathcal{C}_i$ is a strong $(2^i, \alpha \cdot 2^i)$ neighborhood cover for the set of regular vertices in graph $\tilde{H}$, where $\alpha \geq 2048 \log^4 \hat{W}$ is some given approximation factor. We denote $\mathcal{C} = \bigcup_{1 \leq i \leq \lambda} \mathcal{C}_i$.

The contracted graph $\hat{H}^t$ at time t is defined as follows. Its vertex set consists of two subsets: set S of regular vertices (as before, this set contains all regular vertices of C that serve as endpoints of edges in $\hat{E}^t$), and set $\{u(C') \mid C' \in \mathcal{C}\}$ of supernodes. The edge set is defined as follows. For every regular vertex $s \in S$, and supernode $u(C')$, we add an edge $(s, u(C'))$ iff cluster C' contains at least one vertex in $R(s) \cup \{s\}$ (recall that $R(s)$ is the set of all fake copies of s). The length of the edge is 2^i, where i is the index for which $C' \in \mathcal{C}_i$.

Intuitively, we will maintain the neighborhood covers $\mathcal{C}_i$, for $1 \leq i \leq \lambda$ using the algorithmic framework from Section 4.4. Once the collection $\mathcal{C}_i$ of clusters is initialized using algorithm InitNC, it will only undergo allowed updates, which include: DeleteVertex (delete a vertex from a cluster in $\mathcal{C}_i$); AddSuperNode (add a supernode to a cluster in $\mathcal{C}_i$ to reflect a supernode-splitting operation); and ClusterSplit (create a new cluster, that is a sub-graph of an existing cluster in $\mathcal{C}_i$). Once graph $\hat{H}$ is initialized, all subsequent changes to $\hat{H}$, due to the changes in the collections $\{\mathcal{C}_i\}_{i=1}^{\lambda}$ of clusters can be realized via standard update operations. Therefore, we can view the graph $\hat{H}$ as an instance of the $\mathsf{MaintainCluster}$ problem that undergoes a sequence of valid update operations.

The framework from Section 4.4 guarantees (see Theorem 4.11), that for all $1 \le i \le \lambda$, every regular vertex of $\tilde{H}$ lies in at most $W^{O(1/\log\log W)}$ clusters in $\mathcal{C}_i$ over the course of the algorithm, where W is the total number of regular vertices in $\tilde{H}^0$; it is easy to verify that $W \le N^0(C) + |\hat{E}| \le 2N^0(C)$. Moreover, as observed already, for every vertex $s \in S$, $|R(s)| \le \mu$. Therefore, for every vertex $s \in S$, there may be at most $\mu \cdot W^{O(1/\log\log W)}$ clusters in $\bigcup_i \mathcal{C}_i$ that ever contained a vertex of $R(s) \cup \{s\}$, and so the total number of edges that are ever incident to s in $\hat{H}$ is bounded by $\mu \cdot W^{O(1/\log\log W)}$. We can then use $\mu' = \mu \cdot W^{O(1/\log\log W)} \le \mu \cdot (N^0(C))^{O(1/\log\log N^0(C))}$ as the dynamic degree bound for the graph $\hat{H}$. Lastly, observe that $N^0(\hat{H}) \le |\hat{E}^0| \le N^0(C)/\hat{W}^{\epsilon}$. We now turn to prove that the distances between the vertices of S are approximately preserved in $\hat{H}$.

7.3 Distance Preservation in the Contracted Graph

Claim 7.2 *At every time $t \ge 0$, for every pair $s, s' \in S$ of vertices, if $\mathsf{dist}_C(s,s') \le 2048D\log^4\hat{W}$, then $\mathsf{dist}_{\hat{H}}(s,s') \le 4\mathsf{dist}_C(s,s')$.*

Proof: Consider any time point $t \ge 0$, and let $\hat{H}$ be the contracted graph at time t. Let $s, s' \in S$ be a pair of vertices with $\mathsf{dist}_C(s,s') \le 2048D\log^4\hat{W}$, Let P be the shortest s-s' path in graph C, and let $\ell^* \le 2048D\log^4\hat{W}$ be its length. Let $e_1, \ldots, e_q$ be the edges of $\hat{E}$ lying on P, indexed in the order of their appearance. We denote $s_0 = s$, $s_{q+1} = s'$, and, for $1 \le j \le q$, we let $s_j \in S$ be the endpoint of e_j that is a regular vertex. Let $P_1, \ldots, P_{q+1}$ be the sub-paths of P obtained by deleting the edges $e_1, \ldots, e_q$ from P, indexed in the order of their appearance on P. Notice that a path P_j may consist of a single vertex $\hat{s} \in S$. Consider now the graph obtained from P by first deleting the edges $e_1, \ldots, e_q$ from it, and then adding, for all $1 \le j \le q$, the unique fake replacement edge e_j^F of edge e_j to the graph. Then the resulting graph is a subgraph of $\tilde{H}$, and it is a collection of $q+1$ paths $P'_1, \ldots, P'_{q+1}$, where for all $1 \le j \le q$, path P'_j is obtained from P_j by possibly appending one fake edge at the beginning, and possibly appending one fake edge at the end of it. (We note that it is possible that s_j is an endpoint of both e_j and e_{j+1}, and that the fake copies e_j^F, e_{j+1}^F share an endpoint s_j^F; in this case, we view P'_{j-1} as terminating at vertex s_j^F, path P'_j as consisting of only the vertex s_j, and path P'_{j+1} as originating at s_j^F.) Notice that for all $1 \le j \le q$, either the last endpoint of P'_j is s_j and the first endpoint of P'_{j+1} is a fake copy of s_j, or vice versa. Notice also that $P'_1, \ldots, P'_{q+1}$ are paths in graph $\tilde{H}$, and that $\sum_j \ell_{\tilde{H}}(P'_j) = \ell^*$. For all $1 \le j \le q+1$, we denote $\ell_j = \ell_{\tilde{H}}(P'_j)$.

Consider now some index $1 \le j \le q+1$. Let i_j be the smallest integer, so that $\ell_j \le 2^{i_j}$. Since $\ell_j \le \ell^* \le 2048D\log^4\hat{W}$, and $\lambda = \left\lceil \log(2048D\log^4\hat{W}) \right\rceil$, $i_j \le \lambda$ holds. Consider now cluster set $\mathcal{C}_{i_j}$. Since this cluster set is a strong $(2^{i_j}, \alpha \cdot 2^{i_j})$-neighborhood cover for graph $\tilde{H}$, there is some cluster $C_j \in \mathcal{C}_{i_j}$ that contains the two endpoints of path P_j. Therefore, there is an edge $(s_{j-1}, u(C_j))$ and an edge $(u(C_j), s_j)$ in graph $\hat{H}$. We let $\hat{P}_j$ denote the path in graph $\hat{H}$, that is the union of these two edges. As the length of each of the two edges is 2^{i_j}, the length of path $\hat{P}_j$ is $2 \cdot 2^{i_j} \le 4\ell_j$. By concatenating the paths $\hat{P}_1, \ldots, \hat{P}_q$, we obtain a path in graph $\hat{H}$ connecting s to s', of length at most $4\sum_{j=1}^{q+1} \ell_j \le 4\mathsf{dist}_C(s,s')$. □

It is easy to verify that the opposite direction is also true: $\mathsf{dist}_{\hat{H}}(s,s') \le O(\alpha \cdot \mathsf{dist}_C(s,s'))$ for any pair s, s' of vertices in S. We do not use this claim directly, so we do not prove it formally.

8 Putting Everything Together: Proof of Theorem 4.14

In this section we combine all tools that we have developed so far, in order to obtain a proof of Theorem 4.14. In order to do so, we prove the following theorem by induction:

Theorem 8.1 *There is a universal constant $c > 1$, and deterministic algorithm for the* MaintainCluster *problem, that, given a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, where C is a connected graph, undergoing a sequence of valid update operations with dynamic degree bound $\mu \geq 1$, and parameters $c/\log\log((N^0(C))^{2/z}) < \epsilon < 1$, and $\hat{W} \geq N^0(C)\mu$, where $N^0(C)$ is the number of regular vertices in C at the beginning of the algorithm, such that $N^0(C) \leq \hat{W}^{\epsilon z/2}$, for some integer $1 \leq z \leq \lceil 2/\epsilon \rceil$, achieves approximation factor $\alpha_z = (\log \hat{W})^{2c \cdot 2^{2z}/\epsilon^2}$ and has total update time at most:*

$$4^{cz} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + cz}$$

.

Theorem 4.14 follows immediately from Theorem 8.1, by setting $z = \lceil 2/\epsilon \rceil$. The resulting algorithm achieves approximation factor at most $(\log \hat{W})^{2c \cdot 2^{6/\epsilon}/\epsilon^2} = (\log \hat{W})^{2^{O(1/\epsilon)}}$, and it has total update time at most: $N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}$.

The remainder of this section is dedicated to the proof of Theorem 8.1. Throughout the proof, we assume that c is a large enough constant.

The proof is by induction on z. The base case is when $z \leq 10$, and so $N^0(C) \leq \hat{W}^{5\epsilon}$. In this case, we use Algorithm AlgSlow from Theorem 4.15, that achieves approximation factor $O((\log \hat{W})^4) \leq (\log \hat{W})^c$ (assuming that c is a large enough constant), and has total update time $O((N^0(C))^2 \mu D \operatorname{poly}\log \hat{W}) \leq O(N^0(C)\mu D \hat{W}^{5\epsilon} \operatorname{poly}\log \hat{W})$.

From now on we assume that we are given some integer $z > 10$, and that the theorem holds for all integers smaller than z. From Observation 4.16, if $D > \hat{W}$, or $\hat{W} < 2^{O(1/\epsilon^2)}$, then the running time of AlgSlow is at most:

$$O(N^0(C)\mu D^2 \cdot 2^{O(1/\epsilon^2)} \operatorname{poly}\log \hat{W}) \leq O\left(N^0(C)\mu D^2 (\log \hat{W})^{O(1/\epsilon^2)}\right) < 4^c \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2},$$

and it achieves approximation factor $O((\log \hat{W})^4) \leq (\log \hat{W})^c$. Therefore, we assume from now on that $D < \hat{W}$, and that $\hat{W} \geq 2^{\Omega(1/\epsilon^2)}$. In particular, $\hat{W} \geq \max\left\{N^0(C)\mu, |E(C)|, 2^{\Omega(1/\epsilon)}\right\}$ holds throughout the algorithm. The algorithm consists of four phases, that we now describe.

8.1 Phase 1

In this phase, we run the algorithm AlgMaintainGoodCluster from Theorem 6.2 for the MaintainGoodCluster problem on $\mathcal{I}, \Sigma$, with the same parameters ϵ, $\hat{W}$. Recall that, as long as the flag F^b_C is not raised, the algorithm essentially solves the MaintainCluster problem on cluster C, with approximation factor $(\log \hat{W})^{O(1/\epsilon^2)} \leq \alpha_z$. The running time of this algorithm is bounded by:

$$O\left(N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{O(\epsilon)} \cdot (\log \hat{W})^{O(1/\epsilon^2)}\right) \leq \frac{c}{4} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + cz}.$$

Once the algorithm raises flag F^b_C, the first phase terminates. Recall that, once flag F^b_C is raised Algorithm MaintainGoodCluster supplies a bad witness $\hat{E}$ for C, where $\hat{E}$ is $(\hat{D}, \rho)$-pseudocut $\hat{E}$, with $|\hat{E}| < N^0(C)/\hat{W}^\epsilon$; as before, $\hat{D} = 2^{30} D \log^{10} \hat{W}$, and $\rho = \hat{W}^\epsilon$.

8.2 Phase 2

Our algorithm for Phase 2 follows the outline and the tools developed in Section 7, and consists of four parts (or four different algorithms) that we run in parallel. The first algorithm, that we refer to as Alg_1, constructs and maintains the modified graph $\tilde{H}$, and maintains the edge set $\hat{E}$, using the cluster C, the initial bad witness $\hat{E}^0$ for C, and the update sequence Σ for C. This algorithm starts by constructing an initial graph $\tilde{H}$, and then produces an online update sequence $\tilde{\Sigma}$ for it, based on the update sequence Σ for C. The second algorithm, that we call Alg_2, maintains the neighborhood covers $\mathcal{C}_1, \ldots, \mathcal{C}_\lambda$ for graph $\tilde{H}$, and it also maintains the contracted graph $\hat{H}$. This algorithm, after initializing the contracted graph $\hat{H}$, uses the update sequence $\tilde{\Sigma}$ for $\tilde{H}$ and the neighborhood covers $\mathcal{C}_1, \ldots, \mathcal{C}_\lambda$ of graph $\tilde{H}$, in order to produce an online update sequence $\hat{\Sigma}$ for graph $\hat{H}$, that allows us to maintain the graph $\hat{H}$. The third algorithm, that we refer to as Alg_3, simply runs the algorithm for the MaintainCluster problem from the induction hypothesis on graph $\hat{H}$. Lastly, the fourth algorithm, called Alg_4, maintains an ES-Tree in graph C, that is rooted at the vertices of S. We also provide an algorithm for supporting queries short-path-query(C, v, v') in graph C, using the data structures that are maintained these algorithms. We now describe each of these algorithms in turn.

8.2.1 Maintaining the Modified Graph – Algorithm Alg_1

We initialize the set S of regular vertices of C to contain all regular vertices that are endpoints of the edges of $\hat{E}^0$. We also initialize the modified graph $\tilde{H}$. We then use the algorithm from Claim 7.1, that, given the online update sequence $\Sigma = (\sigma_1, \sigma_2, \ldots)$ for C, produces, at each time $t > 0$, a sequence $\tilde{\Sigma}_t$ of valid update operations for graph $\tilde{H}$, such that $\tilde{\Sigma}_t$ contains a constant number of valid update operations, and the length of the description of each operation, as well as the time required to compute $\tilde{\Sigma}_t$, is asymptotically bounded by the length of the description of σ_t. As shown in Claim 7.1, this algorithm ensures that the graph obtained at time t is precisely $\tilde{H}^t$. The algorithm also maintains the dynamic set $\hat{E}$ of edges and the set S of vertices. Lastly, recall that the dynamic degree bound for the dynamic graph $\tilde{H}$ is μ, and that the initial number of regular vertices in $\tilde{H}$, $N^0(H) \leq 2N^0(C)$. We denote this algorithm by Alg_1. The running time of the algorithm is bounded asymptotically by the total length of the description of the update sequence Σ, which is in turn bounded $O(|E^*|)$, where E^* is the total number of edges that are ever present in graph C, so in particular, $|E^*| \leq O(N^0(C)\mu)$. Therefore, the total update time of algorithm Alg_1 is $O(N^0(C)\mu)$.

8.2.2 Maintaining the Neighborhood Covers of $\tilde{H}$ and the Contracted Graph – Algorithm Alg_2

As before, we let $\lambda = \left\lceil \log(2048 D \log^4 \hat{W}) \right\rceil$. We now fix an index $1 \leq i \leq \lambda$, and describe an algorithm for maintaining a strong $(2^i, \alpha_{z-1} \cdot 2^i)$-neighborhood cover $\mathcal{C}_i$ of the regular vertices in the modified graph $\tilde{H}$, as it undergoes a sequence $\tilde{\Sigma}$ of valid update operations. The algorithm follows the framework from Section 4.4. For every vertex $x \in V(\tilde{H})$, we maintain a list $\mathsf{ClusterList}_i(x)$ of all clusters $C' \in \mathcal{C}_i$ containing x, and similarly, for every edge $e \in E(\tilde{H})$, we maintain a list $\mathsf{ClusterList}_i(e)$ of all clusters $C' \in \mathcal{C}_i$ containing e. For every cluster $C' \in \mathcal{C}_i$, we denote by $N^0(C')$ the number of regular vertices that C' contained when it was first added to $\mathcal{C}_i$. We now describe the algorithm for maintaining the neighborhood cover $\mathcal{C}_i$ in more detail.

Algorithm for Maintaining $\mathcal{C}_i$. For convenience, we denote $W = N^0(\tilde{H})$; recall that $W \leq 2N^0(C)$. We also denote $D_i = 2^i$. At the beginning of the algorithm, before any update operations from $\tilde{\Sigma}$ are processed, we apply Procedure InitNC from Section 4.3 to graph $\tilde{H}$, with distance parameter 2^i, and we add to $\mathcal{C}_i$ the resulting collection of clusters, that, from Lemma 4.7, form a strong $(D_i, 1024 D_i \log^4 W)$-neighborhood cover of the set of regular vertices of $\tilde{H}$. The running time of the algorithm, from Lemma 4.7, is $O(N^0(\tilde{H}) \cdot D_i \cdot \mu \cdot \operatorname{poly}\log(N^0(\tilde{H})\mu)$. We need the following observation regarding this initial set $\mathcal{C}_i$ of clusters.

Observation 8.2 *For every cluster $C' \in \mathcal{C}_i$, $N^0(C') \le 2N^0(C)/\hat{W}^{\epsilon} \le \hat{W}^{\epsilon(z-1)/2}$.*

Proof: Let $C' \in \mathcal{C}_i$ be any cluster. Recall that, for every fake edge $e^F = (u, v^F)$, an endpoint v^F of e^F, that is a fake regular vertex, has degree 1 in the initial graph $\tilde{H}^0$. Moreover, the total number of fake regular vertices in $\tilde{H}^0$ is at most $|\hat{E}| \le N^0(C)/\hat{W}^{\epsilon}$. Consider the graph C'', that is obtained from C' after all fake edges are deleted from it. Then C'' is a subgraph of $C \setminus \hat{E}$, and it has diameter at most $2^{i+10}\log^4 W \le 2^{11} \cdot 2^{\lambda} \log^4 \hat{W} < \hat{D}$, since $\lambda = \left\lceil \log(2048 D \log^4 \hat{W}) \right\rceil$, and $\hat{D} = 2^{30} D \log^{10} \hat{W}$.

Since $\hat{E}$ is a $(\hat{D}, \rho)$-pseudocut in C, the number of regular vertices in C'' is at most $N^0(C)/\rho = N^0(C)/\hat{W}^{\epsilon}$. Therefore, altogether, $N^0(C') \le 2N^0(C)/\hat{W}^{\epsilon}$. Since we have assumed that $N^0(C) \le \hat{W}^{\epsilon z/2}$, we get that $N^0(C') \le \hat{W}^{\epsilon(z-1)/2}$. □

As the algorithm progresses, new clusters may be added to $\mathcal{C}_i$, but each such new cluster C'' will always be a subgraph of a cluster that is currently in $\mathcal{C}_i$. Since no new regular vertices are ever added to $\tilde{H}$, this will guarantee that for every cluster C' that is ever added to $\mathcal{C}_i$, $N^0(C') \le \hat{W}^{\epsilon(z-1)/2}$ always holds. Whenever a cluster C' is added to $\mathcal{C}_i$, we initialize the algorithm for the $\mathsf{MaintainCluster}$ problem from the induction hypothesis on it, using the same parameters ϵ and $\hat{W}$ as before (but note that the distance parameter, 2^i, may be as large as $2^{\lambda} > D$). The algorithm then achieves approximation factor α_{z-1}, and running time at most:

$$4^{c(z-1)} \cdot N^0(C') \cdot \mu^2 \cdot D_i^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + c(z-1)}.$$

As graph $\tilde{H}$ undergoes the sequence $\tilde{\Sigma}$ of valid update operations, we update every cluster $C' \in \mathcal{C}_i$ accordingly, in the same way as in the algorithm from Section 4.4. Consider any update operation $\tilde{\sigma}_t \in \Sigma$. If $\tilde{\sigma}_t$ is a deletion of an edge e, then for every cluster $C' \in \mathsf{ClusterList}_i(e)$, we delete e from C' as well. If $\tilde{\sigma}_t$ is a deletion of an isolated vertex x, then for every cluster $C' \in \mathsf{ClusterList}_i(x)$, we delete x from C as well. Assume now that $\tilde{\sigma}_t$ is a supernode splitting operation, applied to a supernode u, and a set $E' \subseteq \delta_{\tilde{H}}(u)$ of its incident edges. For every cluster $C' \in \mathsf{ClusterList}(u)$, we let $E'_{C'} = E' \cap E(C')$. We then update the cluster C' by performing a supernode-splitting operation in it, for vertex u, with edge set $E'_{C'}$. We then add C' to $\mathsf{ClusterList}(u')$ of the newly created supernode u', and also initialize $\mathsf{ClusterList}(e)$ for every edge e that was just added to $\tilde{H}$.

When an algorithm $\mathsf{MaintainCluster}(C')$ raises the flag F_C, for any cluster $C' \in \mathcal{C}_i$, it needs to provide a pair $v, v' \in V(C')$ of regular vertices of C' with $\mathsf{dist}_{C'}(v, v') > 1024 \cdot D_i \log^4 \hat{W} \ge 1024 \cdot D_i \cdot \log^4(2W)$. We then run Procedure $\mathsf{ProcCut}'(C', v, v', D_i)$, obtaining a cluster $C'' \subseteq C'$, and another cluster C''', which is given implicitly, by listing all edges and vertices of $C' \setminus C'''$. We add cluster C'' to $\mathcal{C}_i$, initializing the $\mathsf{MaintainCluster}(C'')$ data structure on it, and update cluster C', by first deleting all edges of $C' \setminus C'''$ from it, and then deleting resulting isolated vertices of $C' \setminus C'''$, so that at the end of this update, $C' = C'''$ holds. The procedure guarantees that either v or v' is deleted from C' at the end of this update sequence. We also update the data structures $\mathsf{ClusterList}(x), \mathsf{ClusterList}(e)$ for all vertices $x \in C''$ and $x \in C' \setminus C'''$, and for all edges $e \in E(C'')$ and $e \in E(C' \setminus C''')$, exactly like in the algorithm from Section 4.4. It is easy to verify that, throughout the algorithm, $\mathcal{C}_i$ is a strong $(2^i, \alpha_{z-1} 2^i)$ neighborhood cover for the set of regular vertices in $\tilde{H}$. Indeed, the algorithm from Section 4.4 (and, in particular, Procedure $\mathsf{ProcCut}$) guarantee that for every regular vertex $v \in V(\tilde{H})$, there is some cluster $C' \in \mathcal{C}_i$ with $B(v, 2^i) \subseteq V(C')$. This property holds immediately after the application of Procedure InitNC. Since valid update operations cannot decrease distances between regular vertices (from Observation 3.1), this property continues to hold after each update operation (if it held before the update operation). From the analysis of Procedure ProcCut, this property also continues to hold after each application of the procedure (if it held before the application of the procedure). Lastly, the fact that the algorithm for the MaintainCluster problem on each cluster $C' \in \mathcal{C}_i$ supports queries short-path-query(C', v, v') with approximation factor α_{z-1} shows that $\mathcal{C}_i$ is indeed a strong $(2^i, \alpha_{z-1} \cdot 2^i)$ neighborhood cover for the set of regular vertices in $\tilde{H}$.

We denote the algorithm that we have described above, that receives as input the modified graph $\tilde{H}$ and the online update sequence $\tilde{\Sigma}$ for it, and maintains the neighborhood covers $\mathcal{C}_1, \ldots, \mathcal{C}_\lambda$ of $\tilde{H}$, by Alg_2. If we exclude the update time for the MaintainCluster algorithm on clusters $C' \in \mathcal{C}_i$, then, from Corollary 4.10, the total running time of Alg_2 is $O(WD_i\mu \operatorname{poly}\log(W\mu)) \leq O(N^0(C)D_i\mu \operatorname{poly}\log \hat{W})$. Additionally, from Claim 4.8, $\sum_{C' \in \mathcal{C}_i} N^0(C') \leq O(W \log W) \leq O(N^0(C) \log \hat{W})$. Therefore, the total running time of all algorithms MaintainCluster(C') for all clusters C' that ever belonged to $\mathcal{C}_i$ is bounded by:

$$4^{c(z-1)} \cdot N^0(C) \cdot \mu^2 \cdot D_i^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + c(z-1)+1}.$$

Summing up over all $1 \leq i \leq \lambda$, and recalling that $2^\lambda \leq 2^{12} D \log^4 \hat{W}$, we get that the total update time of algorithm Alg_2 is at most:

$$4^{c(z-1)} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + c(z-1)+14} \leq 4^{c(z-1)} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + cz}.$$

Maintaining the Contracted Graph. In addition to the above data structures, we maintain the contracted graph $\hat{H}$, as follows. At the beginning of the algorithm, once, for all $1 \leq i \leq \lambda$, cluster set $\mathcal{C}_i$ is initialized using the InitNC procedure, we construct an initial graph $\hat{H}$. Vertex set of $\hat{H}$ consists of the set S or regular vertices – all regular vertices that serve as endpoints of the edges in $\hat{E}^0$, and the set $\{u(C') \mid C' \in \bigcup_i \mathcal{C}_i\}$ of supernodes. There is an edge $(s, u(C'))$ between a regular vertex $s \in S$ and a supernode $u(C')$, where $C' \in \mathcal{C}_i$ for $1 \leq i \leq \lambda$ iff C' contains s or one of its fake copies. The length of the edge is 2^i.

As the algorithm progresses, and clusters in set $\mathcal{C} = \bigcup_i \mathcal{C}_i$ undergo changes, we update the contracted graph $\hat{H}$ by producing a sequence $\hat{\Sigma}$ of valid update operations for it, as follows. Recall that all changes to clusters in $\mathcal{C}$ are allowed changes, and fall into one of the following three categories:

- DeleteVertex(C', x): delete vertex x from cluster C'. If $x \in S$, or x is a fake copy of some vertex $s \in S$, we may need to delete the edge $(s, u(C'))$ from $\hat{H}$. We check whether s or any of its fake copies still lie in C', using the ClusterList(x) data structure for vertices $x \in \{s\} \cup R(s)$, and, if this is not the case, we delete the edge $(s, u(C'))$ from $\hat{H}$. Since there are at most μ fake copies of s, this check can be performed in $O(\mu)$ time.

- AddSuperNode: add a new supernode to an existing cluster $C' \in \mathcal{C}$; no updates to $\hat{H}$ are necessary.

- ClusterSplit: given a cluster $C' \in \mathcal{C}$, and a subgraph $C'' \subseteq C'$, add C'' to $\mathcal{C}$. In this case, we perform a supernode split in graph $\hat{H}$, for supernode $u(C')$. The edge set E' contains, for every vertex $s \in S$ such that $(\{s\} \cup R(S)) \cap V(C'') \neq \emptyset$, the edge $(s, u(C'))$. This update can be executed in time $O(|V(C'')| \cdot \mu)$.

Additionally, when a vertex s is deleted from the vertex set S (which may only happen when s no longer serves as endpoint of any edge in $\hat{E}$), we delete all edges incident to s from $\hat{H}$, and then delete vertex s from $\hat{H}$. Notice that, since s no longer has any edge incident to it in $\hat{E}$, all fake copies of vertex s have been deleted from the graph $\tilde{H}$ already.

It is immediate to verify that the graph $\hat{H}$ that we maintain via the above valid update operations is identical to the contracted graph defined in Section 7.2. Recall that $N^0(\hat{H}) = |S| \leq |\hat{E}| \leq N^0(C)/\hat{W}^{\epsilon} \leq \hat{W}^{\epsilon(z-1)/2}$, and, as we established in Section 7.2, graph $\hat{H}$ has dynamic degree bound $\mu' = \mu \cdot W^{O(1/\log\log W)} \leq \mu \cdot (N^0(C))^{O(1/\log\log N^0(C))}$ (as before, $W = N^0(\tilde{H}) \leq 2N^0(C)$). Let $D' = 8D$ be the distance threshold for the graph $\hat{H}$. It is immediate to extend Algorithm Alg_2 so that, at every time $t > 0$, given an online update sequence $\tilde{\Sigma}_t$ (that is produced by algorithm Alg_1 and corresponds to the update σ_t in the input sequence Σ for cluster C), produces a sequence $\hat{\Sigma}_t$ of valid update operations for graph $\hat{H}$, so that, at any time $t > 0$, the graph that is obtained from the initial graph $\hat{H}$ by applying the update sequence $\hat{\Sigma}_1 \circ \cdots \circ \hat{\Sigma}_t$ to it is precisely $\hat{H}^t$ – that is, the contracted graph $\hat{H}$ at time t. This can be done without increasing the asymptotic running time of Alg_2.

8.2.3 Running MaintainCluster on the Contracted Graph – Algorithm Alg_3

We have now defined a valid input structure $\hat{\mathcal{I}} = (\hat{H}, \{\ell(e)\}_{e\in E(\hat{H})}, D')$, where $D' = 8D$, that undergoes an online sequence $\hat{\Sigma}$ of valid update operations, with dynamic degree bound $\mu' = \mu \cdot (N^0(C))^{O(1/\log\log N^0(C))} \leq \mu(N^0(C))^{\epsilon/2}$ (from the assumption that $\epsilon \geq \Omega(1/\log\log N^0(C))$ in the statement of Theorem 4.14, and the fact that $N^0(\hat{H}) \leq N^0(C)/\hat{W}^{\epsilon}$). From the above discussion, $N^0(\hat{H})\mu' \leq N^0(C)\mu$, and so we still get that $\hat{W} \geq N^0(\hat{H}) \cdot \mu'$. Additionally, $N^0(\hat{H})(\mu')^2 \leq N^0(C)\mu^2$ holds.

We apply the algorithm for the MaintainCluster problem from the induction hypothesis to $\hat{\mathcal{I}}$, with the same parameters $\hat{W}$ and ϵ; we refer to this algorithm as Alg_3. Recall that this algorithm achieves approximation factor α_{z-1}, and has running time:

$$4^{c(z-1)} \cdot N^0(\hat{H}) \cdot (\mu')^2 \cdot (8D)^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + c(z-1)} \leq 4^{c(z-1)} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + cz}.$$

Whenever Algorithm Alg_3 raises a flag $F_{\hat{H}}$, it provides a pair $s, s' \in V(\hat{H})$ of regular vertices, with $\mathsf{dist}_{\hat{H}}(s, s') > 1024D' \log^4 \hat{W} = 2^{13}D \log^4 \hat{W}$. We the raise flag F_C, with the vertices s and s'. Note that from Claim 7.2, $\mathsf{dist}_C(s, s') > 1024D \log^4 \hat{W}$ must hold. We then obtain, as part of the input update sequence Σ for cluster C, a sequence Σ' (the flag lowering sequence) of valid update operations for cluster C, at the end of which either s or s' are deleted from C. This sequence of update operations is processed by Algorithm Alg_1, that produces a sequence $\tilde{\Sigma}'$ of update operations for the modified graph $\tilde{H}$. Update sequence $\tilde{\Sigma}'$ is in turn processed by Algorithm Alg_2, that produces a sequence $\hat{\Sigma}'$ of update operations for the contracted graph $\hat{H}$. Lastly, Algorithm Alg_3 processes the update sequence $\hat{\Sigma}'$ on graph $\hat{H}$. Since either s or s' are deleted from C at the end of sequence Σ', we are guaranteed that either s or s' are deleted from $\hat{H}$ at the end of sequence $\hat{\Sigma}'$. This ensures that, once flag $F_{\hat{H}}$ is raised, algorithm Alg_3 receives, as part of its update sequence, a flag-lowering sequence $\hat{\Sigma}'$ of valid update operations, at the end of which at least one of s, s' is deleted from $\hat{H}$, as required.

8.2.4 The ES-Tree: Algorithm Alg_4

Our last ingerdient is algorithm Alg_4, that maintains a generalized ES-Tree τ from Theorem 3.2, in the graph that is obtained from cluster C, by adding a new source vertex s^*, that connects to every vertex in S with an edge of length 1. The tree τ has depth $2\hat{D}$. As cluster C undergoes valid update operations, we perform the same valid update operations in the data structure τ. When a vertex s is removed from S, we delete the edge (s, s^*) from the tree. Once vertex s^* becomes an isolated vertex in the tree τ, Phase 3 terminates. Observe that the latter can only happen when all vertices are deleted from S, which in turn can only happen when $\hat{E} = \emptyset$ holds.

We maintain a list L of all regular vertices of C that do not lie in the tree τ. Whenever $L \neq \emptyset$, we raise the flag F_C, and supply a pair (s, v) of vertices, where s is any vertex in S, and v is any vertex in L. We are then guaranteed that $\mathsf{dist}_C(s, v) \geq \hat{D} > 1024D \log^4 \hat{W}$.

From Theorem 3.2, the total update time of Algorithm Alg_4 is $\widetilde{O}(N^0(C) \cdot \mu \cdot \hat{D}) \leq N^0(C) \cdot \mu \cdot D \cdot (\log \hat{W})^c$.

Responding to Queries

Suppose we are given a query $\mathsf{short\text{-}path\text{-}query}(C, v, v')$, where v and v' are regular vertices in graph C. Recall that flag F_C must be down when the query is received, and so the list L maintained by Alg_4 is empty, and flag $F_{\hat{H}}$ is down as well. We compute a path P connecting v to v' in C, as follows. First, we use the $\mathsf{ES\text{-}Tree}$ τ to compute a path Q in graph C, connecting v to some vertex $s \in S$, and path Q' in graph C, connecting v' to some vertex $s' \in S$. The time required to compute both paths is proportional the number of edges on them. If $s = s'$, then we obtain a path connecting v to v' in C by concatenating the paths Q and Q'. As both paths have length at most $2\hat{D} \leq O(D \log^{10} \hat{W})$, we obtain a v-v' path of desired length. We assume from now on that $s \neq s'$.

Next, we run query $\mathsf{short\text{-}path\text{-}query}(\hat{H}, s, s')$ in the data structure maintained by Algorithm Alg_3. Recall that the distance threshold for the $\mathsf{MaintainCluster}$ problem on graph $\hat{H}$ is $D' = 8D$, and that the algorithm achieves approximation factor α_{z-1}. Therefore, in response to this query, we obtain a path $\hat{P}$ connecting s to s' in graph $\hat{H}$, whose length is $\hat{\ell} \leq \alpha_{z-1} D' = 8\alpha_{z-1} D$.

We transform path $\hat{P}$ into a path connecting vertex s to vertex s' in graph C, as follows. Denote $\hat{P} = (s_0 = s, u_1, s_1, \ldots, u_{r-1}, s_r = s')$, where $s_0, \ldots, s_r$ are vertices of S, and $u_1, \ldots, u_{r-1}$ are supernodes. Fix an index $1 \leq j < r$, and consider the supernode u_j; assume that $u_j = u(C_j)$, where $C_j \in \mathcal{C}_{i_j}$, for some $1 \leq i_j \leq \lambda$.

We are then guaranteed that either s_{j-1}, or one of its fake copies lie in C_j; in the former case, we set $s'_{j-1} = s_{j-1}$, and in the latter case, we let s'_{j-1} be a fake copy of s_{j-1} lying in C_j. Similarly, we are guaranteed that either s_j or one of its fake copies lie in C_j; in the former case, we set $s'_j = s_j$, and in the latter case, we let s'_j be a fake copy of s_j. Next, we use query $\mathsf{short\text{-}path\text{-}query}(C_j, s'_{j-1}, s'_j)$ to the algorithm for solving the $\mathsf{MaintainCluster}$ problem in graph C_j that is part of Algorithm Alg_2. As the result, we obtain a path P'_j, connecting s'_{j-1} to s'_j in C_j, of length at most $\alpha_{z-1} \cdot 2^{i_j}$. The time required to compute the path P'_j is $O(|E(P'_j)|)$. For every fake regular vertex v^F on path P'_j, we replace v^F with the original regular vertex v (for which v^F is a fake copy). Similarly, every fake edge e^F on P'_j is replaced by its original edge $e \in E(C)$. We then obtain a path P_j in graph C_j, connecting s_{j-1} to s_j, of length at most $\alpha_{z-1} \cdot 2^{i_j}$. Recall that the lengths of the edges (s_{j-1}, u_j) and (s_j, u_j) were 2^{i_j}. By concatenating the paths $P_1, \ldots, P_{r-1}$, we obtain a path P' in graph C, connecting s to s', whose length is at most $\alpha_{z-1} \ell_{\hat{H}}(\hat{P}) \leq 8\alpha_{z-1}^2 D$. The time that the algorithm spent on computing the path P' is $O(|E(P')|)$. Lastly, we let P be the v-v' path in graph C, obtained by concatenating paths Q, P' and Q'. Then path P has length at most:

$$\begin{aligned}8\alpha_{z-1}^2 D + O(D\log^{10}\hat{W}) &\leq 10\alpha_{z-1}^2 D\\ &\leq 10((\log\hat{W})^{2c\cdot 2^{2(z-1)}/\epsilon^2})^2\cdot D\\ &\leq (\log\hat{W})^{2c\cdot 2^{2+2(z-1)}/\epsilon^2}\cdot D\\ &\leq (\log\hat{W})^{2c\cdot 2^{2z}/\epsilon^2}\cdot D\\ &= \alpha_z\cdot D.\end{aligned}$$

The total running time for responding to query $\mathsf{short\text{-}path\text{-}query}(C, v, v')$ is $O(|E(P)|)$.

8.2.5 Running Time of Phase 2

Recall that the running time of Algorithm Alg_1 is $O(N^0(C)\mu)$, the running time of Algorithm Alg_2 is bounded by $4^{c(z-1)}\cdot N^0(C)\cdot\mu^2\cdot D^3\cdot\hat{W}^{c\epsilon}\cdot(\log\hat{W})^{c/\epsilon^2+cz}$, the running time of Algorithm Alg_3 is bounded by $4^{c(z-1)}\cdot N^0(C)\cdot\mu^2\cdot D^3\cdot\hat{W}^{c\epsilon}\cdot(\log\hat{W})^{c/\epsilon^2+cz}$, and the running time of Algorithm Alg_4 is bounded by $N^0(C)\cdot\mu\cdot D\cdot(\log\hat{W})^c$. Therefore, the total running time of the algorithm for Phase 2 is at most:

$$3\cdot 4^{c(z-1)}\cdot N^0(C)\cdot\mu^2\cdot D^3\cdot\hat{W}^{c\epsilon}\cdot(\log\hat{W})^{c/\epsilon^2+cz}$$

8.3 Phase 3

The third phase starts when $S=\emptyset$ holds, and terminates when the number of regular vertices in C falls below $4N^0(C)/\rho$. During this phase, the algorithm will repeatedly raise flag F_C, every time providing a pair v, v' of regular vertices of C with $\mathsf{dist}_C(v, v') > 1024D\log^4\hat{W}$. Therefore, the only updates to cluster C that occur over the course of the phase are due to flag-lowering sequences.

We denote by C^* the cluster C at the beginning of Phase 3. Note that, since Phase 2 has terminated, $\hat{E}=\emptyset$ now holds. We start with the following observation.

Observation 8.3 *For every regular vertex $v^*\in V(C^*)$, the number of regular vertices in $B_{C^*}(v^*, \hat{D})$ is at most $N^0(C)/\rho$.*

Proof: Consider the graph $C' = C\setminus\hat{E}$, as it evolves over the course of the second phase, and let $B = B_{C'}(v^*, \hat{D})$. At the beginning of the second phase, from the definition of a bad witness, the initial ball B contains at most $N^0(C)/\rho$ regular vertices. We claim that no new regular vertices join the set B over the course of the second phase.

In order to prove this, it is enough to consider every update operation $\sigma_t\in\Sigma$ one-by-one, and to show that no new regular vertex joins B as the result of σ_t. Indeed, if σ_t is the deletion of an edge, then this may only increase distances in graph C', so no new vertices join the set B. Similarly, if σ_t is the deletion of an isolated vertex of C, then no new vertices may join B. Assume now that σ_t is a supernode splitting operation of a supernode u, with edge set E'. Recall that, as the result of this operation, we add a new supernode u' to C. For every edge $e=(u,v)\in E'$, if $e\notin\hat{E}$, then edge $e'=(u',v)$, of length $\ell(e')$ is added to C. if $e\in\hat{E}$, then edge $e'=(u',v)$ is added to both C and $\hat{E}$, so it is not added to C'. Therefore, in graph C', we insert a new vertex u', and, for every edge $e=(u,v)\in E'\setminus\hat{E}$, we add edge (u',v) of length $\ell(e)$ to C'. It is easy to see that the distance between any pair of regular vertices in C' may not decrease as the result of this operation, so no new regular vertices may join set B. $\square$

Recall that, from Observation 3.1, valid update operations may not decrease distances between any pair of regular vertices. Therefore, throughout the remainder of the algorithm, as C^* undergoes valid update operations, for every regular vertex $v^* \in V(C^*)$, the number of regular vertices in $B_{C^*}(v^*, \hat{D})$ is always bounded by $N^0(C)/\rho$.

The algorithm in Phase 3 consists of a number of iterations, where in each iteration we will ensure that the number of regular vertices in C reduces by a constant factor. The iterations continue as long as the number of regular vertices in C is at least $4N^0(C)/\rho$. Each iteration consists of two steps. The first step is summarized in the following lemma.

Lemma 8.4 *There is a deterministic algorithm, that, given a graph C' obtained from C^* after a sequence of valid update operations, that contains at least $2N^0(C)/\rho$ regular vertices, computes a collection $\mathcal{B} = \{T_1, \ldots, T_q\}$ of disjoint subsets of regular vertices of C', such that:*

- *for all $1 \leq i \leq q$, $|T_i| \leq N^0(C)/\rho$;*
- *for all $1 \leq i < i' \leq q$, if $v \in T_i$ and $v' \in T_{i'}$, then $\mathsf{dist}_{C'}(v, v') > 1024D \log^4 \hat{W}$; and*
- *$\sum_{i=1}^{q} |T_i| \geq N(C')/2$, where $N(C')$ is the number of regular vertices in C'.*

The running time of the algorithm is $O(|E(C')|) = O(N^0(C)\mu)$.

Proof: The proof uses the standard ball-growing technique. Throughout, we denote $D^* = 2048D \log^4 \hat{W}$. We start with $\mathcal{B} = \emptyset$, and $C'' = C'$, and then perform iterations, where in the ith iteration we add vertex set T_i to $\mathcal{B}$, and we delete some vertices and edges from C''. We also maintain a set T_0 of *discarded regular vertices*, that is initialized to $\emptyset$. We will ensure that the following invariants hold at the end of each iteration i:

P1. for all $1 \leq i' \leq i$, $|T_{i'}| \leq N^0(C)/\rho$;

P2. $|T_0| \leq \sum_{i'=1}^{i} |T_{i'}|$;

P3. for every pair $v, v' \in V(C'') \setminus T_0$ of regular vertices, if $\mathsf{dist}_{C''}(v, v') \geq D^*$, then $\mathsf{dist}_{C'}(v, v') \geq D^*$.

P4. for all $1 \leq i' < i'' \leq i$, $\mathsf{dist}_{C'}(T_{i'}, T_{i''}) \geq D^*$; and

P5. for all $1 \leq i' \leq i$, for all pairs $v \in T_{i'}$, $v' \in V(C'') \setminus T_0$ of regular vertices, $\mathsf{dist}_{C'}(v, v') \geq D^*$.

It is easy to verify that all invariants hold at the beginning of the algorithm. The iterations continue as long as there is at least one regular vertex in $V(C'') \setminus T_0$. We now describe the execution of the ith iteration. We assume that all invariants hold at the beginning of the iteration.

Let v_i be an arbitrary regular vertex in $V(C'') \setminus T_0$. We run Dijkstra's algorithm (that we sometimes call weighted BFS) from vertex v_i in graph C''. For $j \geq 1$, the jth layer Λ_j of the BFS is the set of all vertices whose distance from v_i is between $2(j-1)D^*$ and $2jD^*$, that is: $\Lambda_j = B_{C''}(v_i, 2jD^*) \setminus B_{C''}(v_i, 2(j-1)D^*)$. We denote by N_j the number of regular vertices in Λ_j that do not lie in T_0, and we denote by E_j the set of all edges with both endpoints in $\Lambda_1 \cup \cdots \cup \Lambda_j$. For an index $j > 1$, we say that layer Λ_j is good iff $N_j \leq N_1 + \cdots + N_{j-1}$, and $|E_j \setminus E_{j-1}| \leq |E_{j-1}|$. We need the following simple observation.

Observation 8.5 *There is an index $1 < j \leq 3\log \hat{W}$, such that layer Λ_j is good.*

Proof: Assume otherwise, We say that layer Λ_j is *type-1 bad* if $N_j > N_1 + \cdots + N_{j-1}$, and we say that it is *type-2 bad* if $|E_j \setminus E_{j-1}| > |E_{j-1}|$. Since we have assumed that there are more than $3\log \hat{W}$ bad layers, either at least $\left\lceil \log \hat{W} \right\rceil$ layers are type-1 bad, or at least $\left\lceil 2\log \hat{W} \right\rceil$ layers are type-2 bad. Assume first that the former is true, and let $j_1 < j_2 < \cdots < j_{\lceil \log \hat{W} \rceil}$ be indices of type-1 bad layers. Then for all $2 \leq i \leq \left\lceil \log \hat{W} \right\rceil$, $N_{j_i} > N_1 + \cdots + N_{j_i - 1}$ must hold. Therefore, $N_{j_{\lceil \log \hat{W} \rceil}} > N_1 \cdot 2^{\lceil \log \hat{W} \rceil} \geq \hat{W} \geq N^0(C)$, which is impossible. The proof for the second case, where at least $\left\lceil 2\log \hat{W} \right\rceil$ layers are type-2 bad is almost identical and is omitted here. □

We run Dijkstra's algorithm starting from vertex v_i, until we reach the first index $j > 1$, such that Λ_j is a good layer. We then let T_i contain all regular vertices of $(\Lambda_1 \cup \cdots \cup \Lambda_{j-1}) \setminus T_0$, and we let T_0^i contain all regular vertices of $\Lambda_j \setminus T_0$. From the definition of the good layer, $|T_0^i| \leq |T_i|$. We add the set T_i to $\mathcal{B}$, and we add the vertices of T_0^j to T_0. We also delete from C'' all vertices of $\Lambda_1 \cup \ldots \cup \Lambda_{j-1}$ and all edges that are incident to them. This completes the description of the ith iteration. We now show that all invariants continue to hold.

First, from our discussion, Invariant P2 continues to hold. Also, if Invariants P4 and P5 held at the beginning of the iteration, then Invariant P4 continues to hold.

We denote by C''_0 the cluter C'' at the beginning of the iteration, and by C''_1 the cluster C'' at the end of the iteration. Next, we estabish that Invariant P5 continues to hold. Consider any pair of vertices $v \in T_i$, $v' \in V(C''_1) \setminus T_0$. Then $v \in B_{C''_0}(v_i, 2(j-1)D^*)$, while $v' \notin B_{C''_0}(v_i, 2jD^*)$. Therefore, $\mathsf{dist}_{C''_0}(v, v') \geq D^*$. From Invariant P3, $\mathsf{dist}_{C'}(v, v') \geq D^*$, and so Invariant P5 continues to hold.

In order to establish Invariant P3, consider any pair of regular vertices $v, v' \in V(C''_1) \setminus T_0$, with $\mathsf{dist}_{C''_1}(v, v') \geq D^*$. We claim that $\mathsf{dist}_{C''_0}(v, v') \geq D^*$. Indeed, assume otherwise, and let P be any v-v' path in graph C''_0, whose length is less than D^*. Since $P \not\subseteq C''_0$, it must contain at least one vertex that was deleted in the current iteration, that is, a vertex $x \in \Lambda_1 \cup \cdots \cup \Lambda_{j-1}$. But from the definition of set T_0^i, $v \notin \Lambda_1 \cup \cdots \cup \Lambda_j$. Therefore, $\mathsf{dist}_{C''_0}(x, v) \geq 2D^*$ must hold, contradicting the fact that the length of P in C''_0 is less than D^*. We conclude that $\mathsf{dist}_{C''_0}(v, v') \geq D^*$, and, since Invariant P3 held at the beginning of the iteration, we get that $\mathsf{dist}_{C'}(v, v') \geq D^*$, establishing Invariant P3

It now remains to establish Invariant P1, for which it is enough to show that $|T_i| \leq N^0(C)/\rho$. In order to do so, observe that $T_i \subseteq B_{C''_0}(v_i, 8D^* \log \hat{W})$. Since $D^* = 2048 D \log^4 \hat{W}$, while $\hat{D} = 2^{30} D \log^{10} \hat{W}$, we get that $T_i \subseteq B_{C''_0}(v_i, \hat{D}) \subseteq B_{C'}(v_i, \hat{D})$. From Observation 8.3, and since valid update operations may not decrease distances between regular vertices, it then follows that $|T_i| \leq N^0(C)/\rho$. We conclude that all invariants continue to hold at the end of the current iteration.

Note that the running time of iteration i is $O(|E_j|)$, and that all edges of E_{j-1} are deleted from C'' at the end of the iteration. Since $|E_j| \leq O(|E_{j-1}|)$, the running time of iteration i is asymptotically bounded by the number of edges deleted from C'' in this iteration. Therefore, the total running time of the algorithm is $O(|E(C')|) \leq O(N^0(C)\mu)$. The algorithm terminates once ever regular vertex of C'' lies in T_0. We then output all vertex sets that currently lie in $\mathcal{B}$. From the invariants, it is easy to verify that these vertex sets have all required properties. □

We now proceed to describe Phase 3. The phase is executed as long as the number of regular vertices in graph C remains at least $4N^0(C)/\rho$, and consists of a number of iterations. Each iteration is executed as follows. We start by computing a collection $\mathcal{B}$ of sets of regular vertices, using Lemma 8.4. Next, as long as there are two distinct non-empty sets $T', T'' \in \mathcal{B}$ of vertices, we let $v \in T$, $v' \in T''$ be any pair of vertices. We raise flag F_C with the pair v, v' of vertices. From Lemma 8.4, and from the fact that valid update operations may not decrease distances between pairs of regular vertices, we are guaranteed that $\mathsf{dist}_C(v, v') > 1024D \log^4 \hat{W}$. We then obtain a sequence of valid update operations (flag lowering sequence), after which v or v' must be deleted from C. For every regular vertex v'' that is deleted from C as part of the update sequence, if v'' lies in any set $T''' \in \mathcal{B}$, we delete v'' from set T'''. The iteration terminates once all but at most one set in $\mathcal{B}$ become empty. Recall that at the beginning of the iteration, C contained at least $4N^0(C)/\rho$ regular vertices, and at least half of these vertices lied in the sets of $\mathcal{B}$. Since the remaining set in $\mathcal{B}$ may contain at most $N^0(C)/\rho$ regular vertices, at least a quarter of the regular vertices were deleted from C over the course of the current iteration. Therefore, the total number of iterations in Phase 3 is bounded by $O(\log N^0(C)) \leq O(\log \hat{W})$. Each iteration consists of executing the algorithm from Lemma 8.4, whose running time is $O(N^0(C)\mu)$, and additional work that can be charged to the total number of edges and vertices deleted from C over the course of the iteration. Therefore, the total running time of the algorithm for Phase 3 is $O(N^0(C)\mu \log \hat{W})$. Once Phase 3 terminates, we are guaranteed that the number of regular vertices in graph C is bounded by $4N^0(C)/\rho \leq 4\hat{W}^{\epsilon z/2}/\hat{W}^{\epsilon} \leq \hat{W}^{\epsilon(z-1)/2}$. Since flag F_C is repeatedly raised over the course of Phase 3, no queries may be asked over the course of the phase.

8.4 Phase 4

Recall that, at the beginning of Phase 4, cluster C contains at most $\hat{W}^{\epsilon(z-1)/2}$ regular vertices. We apply the algorithm for the $\mathsf{MaintainCluster}$ problem from the induction hypothesis to cluster C, until the end of the algorithm. Recall that the algorithm achieves approximation factor $\alpha_{z-1} \leq \alpha_z$, and has running time at most $4^{c(z-1)} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + c(z-1)}$.

8.5 Final Accounting of the Running Time

To summarize, the running time of Phase 1 is at most:

$$\frac{c}{4} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + cz};$$

the running time of Phase 2 is at most:

$$3 \cdot 4^{c(z-1)} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + cz};$$

the running time of Phase 3 is at most:

$$O(N^0(C)\mu \log \hat{W});$$

and the running time of Phase 4 is at most:

$$4^{c(z-1)} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + c(z-1)}.$$

Altogether, the total update time of the algorithm is bounded by:

$$4^{cz} \cdot N^0(C) \cdot \mu^2 \cdot D^3 \cdot \hat{W}^{c\epsilon} \cdot (\log \hat{W})^{c/\epsilon^2 + cz}.$$

9 Parameters Used in Sections 6 – 8

Input: valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, undergoing a sequence of valid update operations with dynamic degree bound $\mu \geq 1$, and parameters $0 < \epsilon < 1$, and $\hat{W} \geq N^0(C)\mu$, where $N^0(C)$ is the number of regular vertices in C at the beginning of the algorithm.

New parameters:

- $\hat{D} = 2^{30} D \log^{10} \hat{W}$ – distance parameter;
- $\rho = \hat{W}^{\epsilon}$ – balance parameter for pseudocut;
- $\varphi^* = 1/(\log \hat{W})^{O(1/\epsilon)}$ – expansion of the expander X;
- $\eta = \hat{D} \cdot \hat{W}^{\epsilon} (\log \hat{W})^{O(1/\epsilon)}$ – bound on the congestion of the embedding $\mathcal{P}$ of X.
- $\lambda = \left\lceil \log(2048 D \log^4 \hat{W}) \right\rceil$ – number of distance scales for the neighborhood covers $\mathcal{C}_1, \dots, \mathcal{C}_{\lambda}$.

For a cluster C, we denote by $W(C)$ the number of regular vertices in it.

Type-1 Good Cluser C: $W(C) \leq \hat{W}^{10\epsilon}$.

Good Witness for a type-2 Good Cluster: $(\hat{E}^*, X, \mathcal{P})$, where $\hat{E}^* \subseteq E(C)$ has cardinality $\Omega(W(C)/\hat{W}^{3\epsilon})$, X is a φ^*-expander defined over $\hat{E}^*$, with maximum vertex degree at most $O(\log \hat{W})$, and $\mathcal{P}$ is an embedding of X into $C_{|\hat{E}^*}$ with congestion at most η and path lengths at most $O(\hat{D} \log^2 \hat{W})$.

Bad Witness: $(\hat{D}, \rho)$-pseudocut $\hat{E}$ for C, with $|\hat{E}| < W(C)/\hat{W}^{\epsilon}$.

10 Application: Fast Algorithm for Maximum Multicommodity Flow and Minimum Multicut

In this section, we provide an algorithm for (unweighted) Maximum Multicommodity Flow and Minimum Multicut, proving Theorem 1.3. Recall that in both problems, the input is an undirected n-vertex m-edge graph $G = (V, E)$, and a collection $\mathcal{M} = \{(s_1, t_1), \dots, (s_k, t_k)\}$ of pairs of its vertices, called demand pairs. In the Maximum Multicommodity Flow problem, the goal is to send maximum amount of flow between the demand pairs, such that the total amount of flow traversing any edge is at most 1. We denote by $\mathsf{OPT}_{\mathsf{MCF}}$ the value of the optimal solution to this problem. In the Minimum Multicut problem, the goal is to select a minimum-cardinality subset $E' \subseteq E(G)$ of edges, such that, for all $1 \leq i \leq k$, vertices s_i and t_i lie in different connected components of $G \setminus E'$. We denote by $\mathsf{OPT}_{\mathsf{MM}}$ the value of the optimal solution to Minimum Multicut. We use the standard primal-dual technique-based algorithm of [GK98, Fle00] (see also [Mad10]).

For all $1 \leq i \leq k$, let $\mathcal{P}_i$ be the set of all paths in G connecting s_i to t_i, and let $\mathcal{P} = \bigcup_i \mathcal{P}_i$. We assume that graph G is connected (as otherwise we can solve both problems on each of its connected components separately), so in particular $\mathcal{P} \neq \emptyset$. Below is the standard LP-relaxation of the Maximum Multicommodity Flow problem (denoted by LP$_1$), and its dual (denoted by LP$_2$), which is a relaxation of the Minimum Multicut problem.

$$\begin{array}{lll} \text{LP}_1 & & \\ \text{Max} & \sum_i \sum_{P\in\mathcal{P}_i} f(P) & \\ \text{s.t.} & & \\ & \sum_{i=1}^k \sum_{\substack{P\in\mathcal{P}_i:\\ e\in P}} f(P)\le 1 & \forall e\in E \\ & f(P)\ge 0 & \forall 1\le i\le k, \forall P\in\mathcal{P}_i \end{array}$$	$$\begin{array}{lll} \text{LP}_2 & & \\ \text{Min} & \sum_{e\in E} x_e & \\ \text{s.t.} & & \\ & \sum_{e\in P} x_e\ge 1 & \forall 1\le i\le k, \forall P\in\mathcal{P}_i \\ & x_e\ge 0 & \forall e\in E \end{array}$$

We now show an algorithm that approximately solves both LP_1 and LP_2. Over the course of the algorithm, we maintain lengths x_e on edges $e\in E$, where at the beginning, for every edge $e\in E$, we set $x_e = 1/m$. As the algorithm progresses, we may increase the lengths of the edges. We also set $f(P) = 0$ for every path $P\in\mathcal{P}$. So far we have obtained a feasible solution to LP_1 of value 0, and a (possibly infeasible) solution to LP_2, of value 1. The remainder of the algorithm consists of a number of iterations.

Assume for now, that we are given an oracle $\mathcal{O}$, that, in every iteration, either provides a simple path $P\in\mathcal{P}$, whose length (with respect to current edge lengths x_e) is at most 1, or certifies that every path $P\in\mathcal{P}$ has length at least $1/\alpha$, for some approximation factor $\alpha\ge 1$.

The iterations continue as long as the oracle provides a simple path $P\in\mathcal{P}$ of length at most 1. The jth iteration is executed as follows. Let $P_j\in\mathcal{P}$ be the path provided by the oracle. Then we set $f(P_j) = 1$, and we double the length x_e of every edge $e\in E(P_j)$. Notice that this increases the value of the primal solution by (additive) 1, and it increases the value of the dual solution by at most (additive) 1. Therefore, if we denote by c_1 the cost of the current solution to LP_1, and by c_2 the cost of the current solution to LP_2, then, throughout the algorithm, $c_1\ge c_2-1$ always holds. Since we have assumed that $|\mathcal{P}|\ne\emptyset$, after the first iteration, $c_1\ge 1$, and so $c_2\le 2c_1$ holds for the remainder of the algorithm.

The algorithm terminates when the oracle $\mathcal{O}$ certifies that the length of every path in $\mathcal{P}$ is at least $1/\alpha$. Note that, by setting $x'_e = x_e\cdot\alpha$, we obtain a feasible solution to LP_2, of value at most $\alpha c_2\le 2\alpha c_1$.

The flow values $\{f(P)\}_{P\in\mathcal{P}}$ also provide a solution to LP_1, but that solution may be infeasible, since some edges may carry more than one flow unit. However, since we set, at the beginning, for every edge $e\in E(G)$, $x_e = 1/m$, and since, whenever a path containing e is added to $\mathcal{P}$, we double the length of the edge x_e, it is easy to verify that the total flow that any edge $e\in E(G)$ carries is bounded by $\lceil\log m\rceil$. Let f' be the multicommodity flow obtained by scaling the flow f down by factor $1/\lceil\log m\rceil$. Then f' is a feasible fractional solution to Maximum Multicommodity Flow, of value $c'_1 = c_1/\lceil\log m\rceil$. From the above discussion, $c_2\le 2c_1\le 4c'_1\log m$.

Recall that, from LP-duality, $c'_1\le \mathsf{OPT}_{\mathsf{MCF}} = \mathsf{OPT}_{\text{LP}_1} = \mathsf{OPT}_{\text{LP}_2}\le\alpha c_2$. Therefore, $\mathsf{OPT}_{\mathsf{MCF}}\le\alpha c_2\le O(\alpha\log m)c'_1$ holds. We conclude that we have obtained a solution to the Maximum Multicommodity Flow problem, of value $\Omega(\mathsf{OPT}_{\mathsf{MCF}}/(\alpha\log m)$.

Additionally, $\mathsf{OPT}_{\mathsf{MM}}\ge\mathsf{OPT}_{\text{LP}_2}\ge c'_1\ge\Omega(c_2/\log m)$. Therefore, we have obtained a fractional solution $\{x'_e\}_{e\in E(G)}$ to LP_2, of value $\alpha c_2\le O(\alpha\log m)\mathsf{OPT}_{\mathsf{MM}}$. Our last step is to transform this fractional solution to the Minimum Multicut instance $(G,\mathcal{M})$ into an integral one, using the standard ball-growing technique of [LR99, GVY95]. The resulting deterministic algorithm (that is very similar in nature to our Procedure ProcCut, which in fact was inspired by the algorithm of [LR99, GVY95]), obtains an integral solution to the Minimum Multicut problem instance $(G,\mathcal{M})$, in time $O(|E(G)|)$, of cost $O(\log m)\cdot c$, where $c\le O(\alpha\log m)\mathsf{OPT}_{\mathsf{MM}}$ is the cost of the fractional solution to LP_2.

We conclude that the above algorithm provides an $O(\alpha\log m)$-approximate solution for the

Maximum Multicommodity Flow problem, and an $O(\alpha \log^2 m)$-approximate solution for Minimum Multicut, where α is the approximation factor of the oracle $\mathcal{O}$.

10.1 Implementing the Oracle

We now show an algorithm to efficiently implement the oracle $\mathcal{O}$. One difficulty in implementing it via the algorithm from Theorem 1.2 in a straightforward way is that the algorithm from Theorem 1.2, in response to a query short-path-query(C, s_i, t_i) may return an s_i-t_i path P that is non-simple, and moreover, if we let P' be a simple path obtained from P by removing all cycles, then it is possible that $|E(P)| \gg |E(P')|$. This is a problem because the algorithm spends time $O(|E(P)|)$ in order to process the query, but we will only double the lengths of the edges lying on the path P'. This in turn may result in a running time that is too high overall. Ideally, we would like to ensure that, if the algorithm from Theorem 1.2 returns an s_i-t_i path P that is non-simple, and P' is the corresponding simple path, then $|E(P')|$ is close to $|E(P)|$. We overcome this difficulty as follows. Our algorithm consists of $O(1/\epsilon)$ phases. Denote $m' = \lceil 2m \log m \rceil$. Let $\alpha^* = O\left((\log m)^{2^{O(1/\epsilon)}}\right)$ be the approximation factor that the algorithm from Theorem 1.2 achieves on a graph with m' edges. For $j \geq 0$, let $\alpha_j = (\alpha^*)^{2j}$, and let $L_j = m^{j\epsilon}$. We will ensure that the following invariant holds:

I1. For all $j \geq 0$, at the beginning of Phase $(j+1)$, every path $P \in \mathcal{P}$ whose length is at most $1/\alpha_j$ contains at least L_j edges.

Notice that the invariant clearly holds at the beginning of the first phase. We now describe the execution of the $(j+1)$th phase, for some $j \geq 0$.

Execution of Phase $(j+1)$, for $j \geq 0$. We construct a graph G_j, whose vertex set is $V(G_j) = V(G)$. For every edge $e = (v, v') \in E(G)$, and for every integer $1 \leq i \leq \lceil \log m \rceil$, we add an edge $e_i = (v, v')$ to G_j, of length $\frac{2^i}{m} + \frac{1}{2\alpha^* \cdot \alpha_j \cdot L_{j+1}}$. We call edge e_i *the ith copy of* e. Throughout the algorithm, whenever the length of edge e in graph G doubles, we delete from G_j the lowest-length copy of the edge e. This ensures that, if the length of e in G is $2^i/m$, then every copy of e in G_j has length at least $\frac{2^i}{m} + \frac{1}{2\alpha^* \cdot \alpha_j \cdot L_{j+1}}$. We the initialize the data structure from Theorem 1.2 on this new graph G_j, with target distance threshold $D = 1/(\alpha^* \cdot \alpha_j)$, and we denote by $\mathcal{C}$ the weak $(D, \alpha^* \cdot D)$-neighborhood cover of G_j that the algorithm maintains. (Recall that the definition of the Neighborhood Cover problem requires that the length of every edge is at least 1. In order to achieve this, we need to scale all edge lengths so they become integral, and we need to do the same with the parameter D. As this does not change the problem in any way, we ignore this minor technicality).

We mark every demand pair $(s_i, t_i) \in \mathcal{M}$ as *unexplored.* As the algorithm progresses, we will mark some demand pairs as explored. For each such demand pair (s_i, t_i), we will ensure that the distance, in the current graph G_j, between s_i and t_i, is at least $1/(\alpha^* \cdot \alpha_j)$. We now describe a single iteration.

If every demand pair is marked as explored, then the phase terminates. We are then guaranteed that every path in the current graph G_j, connecting any demand pair $(s_i, t_i) \in \mathcal{M}$ has length at least $1/(\alpha^* \cdot \alpha_j)$ in G_j. We claim that in this case, every path $P \in \mathcal{P}$ whose length is at most $1/\alpha_{j+1}$ (in graph G), contains at least L_{j+1} edges. Indeed, assume otherwise, and let $P \in \mathcal{P}$ be a path connecting some demand pair $(s_i, t_i) \in \mathcal{M}$, that has length $\ell \leq 1/\alpha_{j+1}$ in graph G, and contains fewer than L_{j+1} edges. Let P' be an s_i-t_i path in graph G_j, obtained by taking, for every edge $e \in E(P)$, a copy that has smallest length. Then the length of path P' in graph G_j is bounded by:

$$\ell + \frac{1}{2\alpha^* \cdot \alpha_j} \leq \frac{1}{\alpha_{j+1}} + \frac{1}{2\alpha^* \cdot \alpha_j} \leq \frac{1}{\alpha_j \cdot (\alpha^*)^2} + \frac{1}{2\alpha^* \cdot \alpha_j} < \frac{1}{\alpha^* \cdot \alpha_j},$$

a contradiction to the fact that demand pair s_i-t_i is marked as explored. Therfore, when the phase terminates, Ivariant I1 holds.

Assume now that not every demand pair in $\mathcal{M}$ is marked as explored, and let $(s_i, t_i) \in \mathcal{M}$ be any demand pair that is not marked as explored. Let $C = \mathsf{CoveringCluster}(s_i)$ be the cluster of $\mathcal{C}$ containing $B_{G_j}(s, D)$, that the algorithm from Theorem 1.2 maintains. We start by checking, in time $O(1)$, whether $t_i \in C$. If this is not the case, then we are guaranteed that $\mathsf{dist}_G(s_i, t_i) > D = 1/(\alpha^* \cdot \alpha_j)$. We then mark demand pair (s_i, t_i) as explored, and continue to another unexplored demand pair. Otherwise, if $t_i \in C$, then we run query $\mathsf{short\text{-}path\text{-}query}(C, s_i, t_i)$ in the data structure maintained by the algorithm from Theorem 1.2. The algorithm is then guaranteed to return a path connecting s_i to t_i in graph G_j, of length at most $1/\alpha_j$. We denote this path by P. From the way we set the lengths of the edges in graph G_j, we are guaranteed that $|E(P)| \leq 2\alpha^* \cdot L_{j+1}$. Path P immediately gives us the corresponding (possibly non-simple) path P' in graph G, whose length is at most $1/\alpha_j$. Let P'' be a simple path that is obtained from P', after removing all cycles from it. Note that we can compute P'' in time $O(|E(P)|)$, and the query time $\mathsf{short\text{-}path\text{-}query}(C, s_i, t_i)$ also took time $O(|E(P)|)$. Then the length of path P' is bounded by $1/\alpha_j$, and, from Invariant I1, path P'' contains at least L_j edges. We then return the path P'' and terminate the iteration.

The algorithm terminates after $t = \lceil 1/\epsilon \rceil$ phases, at which time we are guaranteed, from Invariant I1, that every path in $\mathcal{P}$ has length at least $1/\alpha_t$, for $\alpha_t = (\alpha^*)^{O(1/\epsilon)} = O\left((\log m)^{2^{O(1/\epsilon)}}\right)$. We denote $\alpha = \alpha_t$, the approximation factor of the oracle $\mathcal{O}$. We now analyze the running time of a single phase.

The time required to maintain the data structure from Theorem 1.2 is $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)}\right)$. The time needed to process every query $\mathsf{short\text{-}path\text{-}query}(C, s_i, t_i)$ is $O(|E(P)|)$, where P is the returned path. Recall that we have established that P contains at most $2\alpha^* \cdot L_{j+1}$ edges, while its corresponding simple path P'' contains at least L_j edges. Therefore, $|E(P)| \leq 2\alpha^* \cdot m^{\epsilon} |E(P'')|$. We charge every edge on path P'' for at most $2\alpha^* \cdot m^{\epsilon}$ edges on path P. Since we double the length of every edge on path P'' in graph G, and since the length of every edge may only be doubled $O(\log m)$ times, an edge of G may be charged at most $O(\log m)$ times over the course of a single phase. Therefore, the total time for processing all queries $\mathsf{short\text{-}path\text{-}query}(C, s_i, t_i)$ over the course of the phase, and also for computing the corresponding simple paths, is bounded by $O\left(m^{1+\epsilon}(\log m)^{2^{O(1/\epsilon)}}\right)$. Lastly, for every demand pair (s_i, t_i), we may spend additional $O(1)$ time in the iteration in which the pair is marked as explored. Therefore, the total running time of a single phase is bounded by $O\left(m^{1+O(\epsilon)}(\log m)^{2^{O(1/\epsilon)}} + k\right)$. Since the total number of phases is bounded by $O(1/\epsilon)$, the total running time of the algorithm implementing the oracle is $O\left(m^{1+O(\epsilon)}(\log m)^{2^{O(1/\epsilon)}} + k/\epsilon\right)$. The time required in order to implement the remainder of the algorithm (that is, updating the flow f and the edge lengths) is subsumed by this running time. Therefore, the total running time of the algorithm is $O\left(m^{1+O(\epsilon)}(\log m)^{2^{O(1/\epsilon)}} + k/\epsilon\right)$. Since this implementation of the oracle achieves approximation factor $\alpha = O\left((\log m)^{2^{O(1/\epsilon)}}\right)$, the final approximation factor that we achieve for both Maximum Multicommodity Flow and Minimum Multicut is $O\left((\log m)^{2^{O(1/\epsilon)}}\right)$.

11 Acknowledgement

We thank Merav Parter for introducing us to the notion of neighborhood covers and to the work of [AP90, ABCP98], that was the inspiration for this work.

A Proofs Omitted from Section 2

A.1 Short Paths in Expanders – Proof of Observation 2.1

The proof follows the standard ball-growing technique. For all $i \geq 1$, we denote $S_i = B_G(v, i)$, and $S'_i = B_G(u, i)$. Consider now some index i, and assume that $|S_i| \leq n/2$. Let $R_i = S_{i+1} \setminus S_i$ be the set of all vertices v' whose distance from v is exactly $i + 1$. Then, since graph G is a φ-expander, $|E_G(S_i, R_i)| = |E_G(S_i, V(G) \setminus S_i)| \geq \varphi|S_i|$. On the other hand, since every vertex in R_i has degree at most Δ, $|E_G(S_i, R_i)| \leq \Delta|R_i|$. Therefore, $|R_i| \geq \varphi|S_i|/\Delta$ must hold. Since $S_{i+1} = S_i \cup R_i$, we get that $S_{i+1} \geq (1 + \varphi/\Delta)|S_i|$. We conclude that, for $q = \lceil \Delta \log n/\varphi \rceil$, $|S_q| \geq n/2$ must hold. Using similar reasoning, $|S'_q| \geq n/2$ must hold, and so there must be a path connecting u to v, whose length is at most $2q + 1 \leq 8\Delta \log n/\varphi$.

A.2 Expander Pruning – Proof of Theorem 2.2

For the sake of the proof, we need to define the notions of volumes of vertex sets, and graph conductance. Given a graph $G = (V, E)$, and a subset $S \subseteq V$ of its vertices, the *volume* of S in G is $\mathrm{Vol}_G(S) = \sum_{v \in S} \deg_G(v)$. Given a cut (A, B) in graph G, its *conductance* is $\psi_G(A, B) = \frac{|E_G(A,B)|}{\min\{\mathrm{Vol}_G(A), \mathrm{Vol}_G(B)\}}$. The *conductance of a graph* G, denoted by $\Psi(G)$, is the minimum conductance of any cut in G. The following observation is immediate from the definitions of graph expansion and conductance.

Observation A.1 *Let $G = (V, E)$ be a connected graph with maximum vertex degree Δ, and let (A, B) be any cut in G. Then:*

$$\frac{\varphi_G(A, B)}{\Delta} \leq \psi_G(A, B) \leq \varphi_G(A, B).$$

In particular:

$$\frac{\Phi(G)}{\Delta} \leq \Psi(G) \leq \Phi(G).$$

In order to prove Theorem 2.2, we use the following theorem from [SW19]

Theorem A.2 (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 $0 < \psi \leq 1$, and a sequence $\Sigma = (e_1, e_2, \ldots, e_k)$ of $k \leq \psi m/10$ 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 $1 \leq i \leq k$:*

- *$S_{i-1} \subseteq S_i$;*
- *$\mathrm{Vol}_G(S_i) \leq 8i/\psi$;*
- *$|E(S_i, V \setminus S_i)| \leq 4i$; and*
- *if $\Psi(G) \geq \psi$, then the conductance of the graph $G_i[V \setminus S_i]$ is at least $\psi/6$.*

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

We are now ready to prove Theorem 2.2. Let $G = (V, E)$ be the input graph, that is a φ-expander, with maximum vertex degree at most Δ. From the definition of expanders, G must be a connected graph, and, from Observation A.1, if we denote by $\psi = \Psi(G)$, then $\psi \geq \varphi/\Delta$. From the statement of Theorem 2.2, the number of edges in the deletion sequence Σ is $k \leq \frac{\varphi|E|}{10\Delta} \leq \frac{\psi|E|}{10}$, as required. We can now apply the algorithm from Theorem A.2 to graph G and edge deletion sequence Σ. We are then guaranteed that for all $1 \leq i \leq k$, $S_{i-1} \subseteq S_i$, as required. Moreover, we are guaranteed that for each resulting graph $G'_i = G_i[V \setminus S_i]$, the conductance of G'_i is at least $\psi/6$, and so, from Observation A.1, we get that $\Phi(G'_i) \geq \Psi(G'_i) \geq \psi/6 \geq \varphi/(6\Delta)$. Therefore, graph G'_i is a $\varphi/(6\Delta)$-expander. We are also guaranteed that $|E(S_i, V \setminus S_i)| \leq 4i$. Lastly, since graph G is connected, for every vertex set S, $|S| \leq \mathrm{Vol}_G(S)$ must hold, and so for all i, $|S_i| \leq \mathrm{Vol}_G(S_i) \leq 8i/\psi \leq 8i\Delta/\varphi$. The running time of the algorithm is $O(k \log |E|)/\psi^2 = \widetilde{O}(k\Delta^2/\varphi^2)$.

B Generalized Even-Shiloach Trees – Proof of Theorem 3.2

Data Structures. We maintain the graph H as an adjacency list: for every vertex v, we maintain a linked list of its neighbors. Throughout the algorithm, we will maintain the following data structures:

- A shortest-path tree T rooted at vertex s, that contains all vertices $x \in V(H)$ with $\mathsf{dist}_H(s, x) \leq D^*$. For every such vertex x, its correct distance $\lambda(x) = \mathsf{dist}_H(s, x)$ from s is stored together with x.
- For every vertex $x \in V(T)$, let $\mathsf{pred}(x)$ be the set of all vertices $y \in V(T)$ with $(x, y) \in E(H)$. We maintain a heap $\mathsf{Heap}(x)$ in which all elements of $\mathsf{pred}(x)$ are stored, where the key associated with each element $y \in \mathsf{pred}(x)$ is $\lambda(y) + \ell(y, x)$.
- For every edge $e = (y, x)$ with $y, x \in V(T)$, we store, together with vertex y, a pointer to its corresponding element in the heap $\mathsf{Heap}(x)$ and vice versa.

Throughout, we denote by m' the total number of edges that are ever present in H, so $m' \leq N^0 \cdot \mu$.

Initialization. We run Dijkstra's algorithm on graph H time $\widetilde{O}(|E(H)|) \leq \widetilde{O}(m')$, up to distance threshold D^*, to construct the initial tree T. For every vertex $x \in V(T)$, we initialize $\lambda(x) = \mathsf{dist}_H(s, v)$, and for all $x \notin V(T)$, we set $\lambda(x) = \infty$. We then process all vertices of $V(T)$ in the order of their distance from s, and insert each such vertex x into all heaps $\mathsf{Heap}(y)$ of all vertices $y \in V(H)$ that are neighbors of x in H. Clearly, initialization takes time at most $\widetilde{O}(m')$.

We now assume that we are given a valid data structure, and describe algorithms for handling updates.

Edge Deletions. The procedure for processing edge deletions is completely standard. The description provided here is due to Chechik [Che18]. It is somewhat different but equivalent to the standard description. Suppose an edge $e = (x, y)$ is deleted from the graph H. We start by updating the heaps $\mathsf{Heap}(x)$, $\mathsf{Heap}(y)$ with the deletion of this edge. If $e \notin E(T)$, then no further updates are necessary.

Therefore, we assume from now on that $e = (x, y)$ is an edge of the tree T, and we assume w.l.o.g. that y is the parent of x in T. Let S contain all vertices of T that lie in the subtree T_x of x. We delete the edge (x, y) from T, thereby disconnecting all vertices of S from T. The remainder of the algorithm consists of two phases. In the first phase, we identify the set $R \subseteq S$ of all vertices, whose distance from s has increased, and connect all remaining vertices of S to T. In the second phase, we attempt to reconnect vertices of R to T.

In order to implement the first phase, we maintain a heap Q of all vertices that need to be examined, where the key associated with each vertex a in Q is $\lambda(a)$. We initialize Q to maintain a single vertex – the vertex x, and we also initialize $R = \emptyset$. Heap Q will have the property that, if some vertex a belongs to Q, and vertex a' was the parent of a in T, then a' was added to R (or, if $a = x$, then $a' \in T$). Moreover, if a is a vertex of Q with smallest key $\lambda(a)$, and a' is the element lying at the top of $\mathsf{Heap}(a)$, then $a' \in T$ must hold. Both these invariants hold at the beginning.

The algorithm iterates, as long as $Q \neq \emptyset$. Let a be a vertex of Q with smallest key $\lambda(a)$. Let b be the element lying at the top of $\mathsf{Heap}(a)$. We check whether connecting a to the tree T via vertex b will allow us to keep $\lambda(a)$ unchanged, or, equivalently, whether $\lambda(a) = \lambda(b) + \ell(a, b)$. If this is the case, then we connect a to the tree T via b, delete a from Q, and proceed to the next iteration. Otherwise, we are guaranteed that $\lambda(a)$ must increase. We then add a to R, add all children of a in the original tree T to the heap Q, and delete a from $\mathsf{Heap}(b')$ for all neighbors b' of a.

Note that the algorithm examines vertices of S in the non-decreasing order of their label $\lambda(a)$ (except that, when some vertex a is reconnected to the tree T, then its descendants will never be examined). A vertex a that is examined is either connected to the tree, or it is added to R, and all its children are added to Q. This ensures that, throughout the algorithm, if a is a vertex of Q with smallest key $\lambda(a)$, and a' is the element lying at the top of $\mathsf{Heap}(a)$, then $a' \in T$ must hold. Indeed, if $a' \notin T$, then we should have examined a' before, and, if it was added to R, then it would have been deleted from $\mathsf{Heap}(a)$.

The first phase terminates once $Q = \emptyset$. It is not hard to see, using standard analysis, that its running time is bounded by $\widetilde{O}(\sum_{x \in R} \deg_H(x))$.

In the second phase, we run Dijkstra's algorithm on the vertices in R, up to distance D^*, trying to reconnect them to the tree T. We also update the heaps of all vertices of R, and of their neighbors, accordingly. This step can also be implemented in time $\widetilde{O}(\sum_{x \in R} \deg_H(x))$.

To summarize, in edge-update operations, whenever, for any vertex x, its distance from s increases, we may need to pay $\widetilde{O}(\deg_H(x))$ in running time. It is then easy to see that the total update time due to edge deletions is bounded by $\widetilde{O}(m'D^*)$.

Deletion of Isolated Vertices. Deletion of isolated vertices is straightforward, and takes $O(1)$ time per vertex. Note that an isolated vertex may not belong to T (unless that vertex is s and T only contains the vertex s), so apart from deleting the vertex from H, no further updates are necessary.

Supernode Splitting. Recall that in a supernode-splitting update, we are given a supernode $u \in U$, and a set $E' \subseteq \delta_H(e)$ of edges. We need to add a new supernode u' to the graph, and, for every edge $e = (u, v') \in E'$, insert an edge $e' = (u', v')$ of length $\ell(e)$ into H.

If $u \notin T$, then we simply set $\lambda(u') = \lambda(u') = \infty$ and terminate the update algorithm.

Assume now that $u \in T$, and assume first that $u \neq s$. Let v be the parent of the vertex u in the tree T. We now proceed as follows:

1. Add a new vertex u' as the child of v to the tree T (it is convenient to think of it as a copy of u); set $\ell(u', v) = \ell(u, v)$, $\lambda(u') = \lambda(u)$, add v to $\mathsf{Heap}(u')$, and add u' to $\mathsf{Heap}(v)$.
2. For every edge $e = (u, v') \in E'$ with $v' \neq v$, add an edge $e' = (u', v')$ of length $\ell(e)$ to the graph H, add v' to $\mathsf{Heap}(u')$, and add u' to $\mathsf{Heap}(v')$. Notice that the insertion of these edges does not decrease the distance of any vertex from s, since $\lambda(u) = \lambda(u')$, and every regular vertex that serves as endpoint to a newly inserted edge is also a neighbor of u.

3. If edge $(u, v) \notin E'$, delete the edge (v, u') from the graph H, and from the ES-Tree data structure, using the edge-deletion update operation.

If $u = s$, then the update procedure is almost identical, except that we initially insert u' as a child of u, and set the length of the edge (u, u') to be 1/2. This ensures that, as edges corresponding to the edge set E' are inserted into the graph, the distances from vertices of H to s do not decrease, and, since the lengths of all edges in H are at least 1, T remains a valid shortest-path tree. In the last step, we delete the edge (u, u') from our data structure using the edge-deletion update operation.

The processing time of this update procedure, excluding the calls to the edge-deletion updates in the ES-Tree data structure, is $\widetilde{O}(|E'|)$. The possible insertion of edge (u, u'), or edge (u, v), if this edge does not lie in E', increases the total number of edges inserted into our data structure. However, since $|E'| \geq 1$ must hold, this new inserted edge can be charged to some edge of E', increasing the total number of edges that are ever present in our data structure by at most factor 2. From the above discussion, if we denote the length of the input update sequence Σ by k, the total update time of the ES-Tree data structure is bounded by $\widetilde{O}(m'D^* + k) \leq \widetilde{O}(N^0\mu D^* + k)$.

Lastly, observe that each update operation in Σ either inserts at least one edge into H, or deletes at least one vertex or an edge from H. Therefore, $k \leq O(m')$, and the total running time of the algorithm is at most $\widetilde{O}(N^0\mu D^*)$.

Responding to Queries SSSP-query**.** Recall that in SSSP-query, we are given a vertex $x \in V(H)$, and our goal is to either correctly establish, in time $O(1)$, that $\mathsf{dist}_H(s, x) > D^*$, or to return a shortest s-x path P, in time $O(|E(P)|)$. Given a query vertex x, we check whether $\lambda(x) = \infty$. If so, we report that $\mathsf{dist}_H(s, x) > D^*$. Otherwise, we retrace the unique path P connecting x to s in the tree T, and return it, in time $O(|E(P)|)$.

C Recursive Composition of RecDynNC Instances – Proof of Lemma 3.5

If $\epsilon \geq c/\log\log W$ for some constant c, then, since we can assume that $\tilde{c}$ is a large enough constant, the desired approximation factor $\alpha_i = (\log W)^{\tilde{c}i\cdot 2^{\tilde{c}/\epsilon}} > W^3$. In this case, we can simply use a known deterministic algorithm for fully dynamic minimum spanning forest of [HdLT01], that has total update time $O(W \log^4 W)$. We simply let $\mathcal{C}$ be the set of all connected components of the graph H. It is easy to verify that $\mathcal{C}$ can be maintained using allowed changes only, in the same asymptotic total update time. Queries short-path-query(C, v, v') can be easily handled by returning the unique v-v' path in the minimum spanning tree of C. Therefore, we assume from now on that $\epsilon < c/\log\log W$.

The proof is by induction on i. The base case is when $i = 1$, so $D \leq 6W^\epsilon$. We use the algorithm from Theorem 3.3, whose approximation factor is $(\log(W\mu))^{2^{O(1/\epsilon)}} \leq \alpha_1$ (if constant $\tilde{c}$ is large enough), and running time is bounded by:

$O\left(W^{1+O(\epsilon)} \cdot D^3 \cdot (\log W)^{O(1/\epsilon^2)}\right) \leq \left(\tilde{c} \cdot W^{1+\tilde{c}\epsilon} \cdot (\log W)^{\tilde{c}/\epsilon^2}\right)$, as required.

For the induction step, we consider some integer $1 < i \leq \lceil 1/\epsilon \rceil$, and assume that the lemma holds for integers below i. Consider the input graph H and the distance threshold $D \leq 6W^{\epsilon i}$. We use a parameter $D' = \lfloor W^{\epsilon(i-1)}/4 \rfloor$. Clearly, $D' \geq W^{\epsilon(i-1)}/8$. We say that an edge e of H is *long* if $\ell(e) > D'$, and we say that it is *short* otherwise.

Let H' be a dynamic graph that is obtained from H by deleting all long edges from it. We can generate a valid update sequence for graph H' from the valid update input sequence Σ for graph H in a natural way, by ignoring updates concerning long edges. Therefore, we now obtained a valid input structure $\mathcal{I}' = \left(H', \{\ell(e)\}_{e\in E(H')}, 4D'\right)$, that undergoes a sequence Σ' of edge-deletion and isolated vertex-deletion operations, with dynamic degree bound 2, and the initial number of the regular vertices in H' is at most W. Since $4D' \leq W^{\epsilon(i-1)}$, we can apply the induction hypothesis to input structure $\mathcal{I}'$, sequence Σ' of update operations, distance bound $4D'$, and approximation factor $\alpha_{i-1} = (\log W)^{\tilde{c}(i-1)\cdot 2^{\tilde{c}/\epsilon}}$. The total update time of this algorithm is bounded by $\left(\tilde{c}^{i-1} \cdot W^{1+\tilde{c}\epsilon} \cdot (\log W)^{\tilde{c}/\epsilon^2}\right)$. We are also guaranteed that for every regular vertex v of H', the total number of clusters in $\mathcal{C}'$ that ever contain v is at most $W^{O(1/\log\log W)}$. We denote by $\mathcal{D}(H')$ the corresponding data structure, by $\mathcal{C}'$ the collection of clusters that the algorithm maintains, and by $\mathcal{U}' = \{V(C) \mid C \in \mathcal{C}'\}$ the corresponding vertex subsets. We denote the algorithm that we have described so far, that maintains the data structure $\mathcal{D}(H')$, by Alg_1.

We use the neighborhood cover $\mathcal{C}'$ in order to define another dynamic graph $\hat{H}$, as follows. We start with letting $\hat{H} = H$, and then round all edge lengths up to the next integral multiple of D', denoting the resulting new length of each edge e by $\hat{\ell}(e)$. Notice that, if e is a short edge, then $\ell(e) \leq \hat{\ell}(e) = D'$, and if e is a long edge, then $\ell(e) \leq \hat{\ell}(e) \leq 2\ell(e)$. Additionally, for every cluster $C \in \mathcal{C}'$, we add a supernode $u(C)$ to graph $\hat{H}$, and connect $u(C)$ with an edge to every **regular** vertex $v \in V(H')$ that lies in C; the length of this edge is $4D'$. This ensures that the length of every edge in $\hat{H}$ is an integral multiple of D'. For each time point t, we denote by $\hat{H}^t$ the graph $\hat{H}$ obtained immediately after the tth update in sequence Σ to graph H is processed.

We now proceed as follows. First, we show an algorithm to construct an initial graph $\hat{H}^0$, and an algorithm to produce an online sequence of valid update operations $\hat{\Sigma}$ for this graph, so that, at every time t (that is, after tth update in sequence Σ for graph H is processed by our algorithm), the resulting graph that we obtain is precisely $\hat{H}^t$. We will also show that, for every pair of regular vertices $v, v' \in V(H)$, the distance between them in $\hat{H}$ is close to that in H. We will use the algorithm from Theorem 3.3 recursively on graph $\hat{H}$, to maintain a neighborhood cover $\hat{\mathcal{C}}$ of regular vertices in graph $\hat{H}$. Lastly, we show that we can use the resulting dynamic neighborhood cover $\hat{C}$ for graph $\hat{H}$ in order to maintain the desired neighborhood cover for graph H.

Maintaining Graph $\hat{H}$. Recall that, from the definition of the RecDynNC problem, initially, the cluster set $\mathcal{C}'$ consists of a single cluster H', and its corresponding collection $\mathcal{U}'$ of vertex subsets contains a single vertex set, $V(H')$. As the time progresses, vertex sets in $\mathcal{U}'$ may only be updated via allowed change operations: DeleteVertex, AddSuperNode, and ClusterSplit. We define the initial graph $\hat{H}^0$ as follows. We set $\hat{H}^0 = H$, except that we set the edge lengths $\left\{\hat{\ell}(e)\right\}_{e\in E(\hat{H}^0)}$ as described above. Additionally, we add a single supernode $u(H')$, that connects to every regular vertex of H' with an edge of length $4D'$. We use the following claim in order to produce input update sequence $\hat{\Sigma}$ for $\hat{H}$.

Claim C.1 *There is a deterministic algorithm that, given, at each time $t \geq 1$, the t-th update σ_t in the input update sequence Σ for $\mathcal{I}$, produces a sequence $\hat{\Sigma}_t$ of valid update operations for graph $\hat{H}$, such that, for all $t \geq 0$, the graph obtained from $\hat{H}^0$ by applying the update sequence $(\hat{\Sigma}_1 \circ \cdots \circ \hat{\Sigma}_t)$ to it is precisely $\hat{H}^t$. The dynamic degree bound for the resulting dynamic graph $\hat{H}$ is at most $W^{O(1/\log\log W)}$. The total update time of the algorithm is bounded by $O\left(\tilde{c}^{i-1} \cdot W^{1+\tilde{c}\epsilon} \cdot (\log W)^{\tilde{c}/\epsilon^2}\right)$.*

Proof: We run Algorithm Alg_1, that solves the RecDynNC problem on graph input structure $\mathcal{I}' = \left(H', \{\ell(e)\}_{e\in E(H')}, 4D'\right)$, that undergoes a sequence Σ' of edge-deletion and isolated vertex-deletion operations, with dynamic degree bound 2, with the initial number of the regular vertices in H' bounded by W. Recall that the algorithm maintains the neighborhood cover $\mathcal{C}'$, achieves approximation factor

$\alpha_{i-1} = (\log W)^{\tilde{c}(i-1)\cdot 2^{\tilde{c}/\epsilon}}$, and has running time at most $\left(\tilde{c}^{i-1} \cdot W^{1+\tilde{c}\epsilon} \cdot (\log W)^{\tilde{c}/\epsilon^2}\right)$. We are also guaranteed that for every regular vertex v of H', the total number of clusters in $\mathcal{C}'$ that ever contain v is at most $W^{O(1/\log\log W)}$.

Recall that initially, $\mathcal{C}' = \{H'\}$. Consider now some time $t > 0$, when an update $\sigma_t \in \Sigma$ for input structure $\mathcal{I}$ arrives. We initialize $\hat{\Sigma}_t = \emptyset$, and we start by updating the data structure $\mathcal{D}(H')$ with the update operation σ_t, which may result in some changes to the vertex sets in $\mathcal{U}' = \{V(C) \mid C \in \mathcal{C}'\}$. We consider the resulting changes to vertex sets of $\mathcal{U}'$ one by one. If the change is AddSuperNode(S, u), for some vertex set $S \in \mathcal{U}'$, then we ignore this update. If the change is DeleteVertex(S, x), where a vertex $x \in S$ is deleted from vertex set $S \in \mathcal{U}'$, and x is a regular vertex, then we add to the sequence $\hat{\Sigma}_t$ an edge-deletion operation for the edge $(x, u(C))$, where C is the cluster of $\mathcal{C}'$ with $V(C) = S$; if x is a supernode then we ignore this change. If the change is ClusterSplit(S, S'), where $S \in \mathcal{U}'$ and $S' \subseteq S$ is a new vertex set that is added to $\mathcal{U}'$, then we add a supernode split operation to $\hat{\Sigma}_t$, defined as follows. Let C be the cluster of $\mathcal{C}'$ with $V(C) = S$. The supernode split operation is performed on supernode $u(C)$, and the corresponding edge set E' contains all edges $(u(C), v)$, where v is a regular vertex of H lying in S'. The new supernode that is added to the graph is $u(C')$, where C' is the new cluster with $V(C') = S'$. Note that the time that is needed in order to compute the set E' of edges is $|S'|$; since the algorithm for the $\mathsf{RecDynNC}$ problem on $\mathcal{I}'$ needs to add vertex set S' to $\mathcal{U}'$, this time is subsumed by the time that Algorithm Alg_1 takes in order to execute the ClusterSplit(S, S') operation.

Lastly, we consider the update operation σ_t itself, and add additional updates to $\hat{\Sigma}_t$ as follows. If σ_t is the deletion of an isolated vertex x from graph H, then x must also currently be an isolated vertex of $\hat{H}$ (as it was just deleted from all clusters containing it). We then add isolated vertex deletion operation for vertex x to $\hat{\Sigma}_t$. If σ_t is the deletion of an edge e, then add to $\hat{\Sigma}_t$ the deletion of e. This completes the description of the algorithm for producing the sequence $\hat{\Sigma}_t$.

It is immediate to verify that for all $t > 0$, the graph obtained by applying update sequence $(\hat{\Sigma}_1 \circ \cdots \circ \hat{\Sigma}_t)$ to graph $\hat{H}^0$ is precisely $\hat{H}^t$. Next, we bound the dynamic vertex degree for the resulting dynamic graph $\hat{H}$. Recall that, from the statement of Lemma 3.5, for every regular vertex v of graph H', the total number of clusters in $\mathcal{C}'$ that ever contain v is bounded by $W^{O(1/\log\log W)}$. Since the inital degree of every regular vertex in H' is at most 2, and no supernode splitting operations are allowed in H, we get that the dynamic degree bound for $\hat{H}$ is $W^{O(1/\log\log W)}$.

The total update time of this algorithm is subsumed by the total update time of Algorithm Alg_1, and is bounded by $O\left(\tilde{c}^{i-1} \cdot W^{1+\tilde{c}\epsilon} \cdot (\log W)^{\tilde{c}/\epsilon^2}\right)$. □

Distance Preservation. We now show that distances between regular vertices in graph $\hat{H}$ are not much larger than the corresponding distances in graph H.

Lemma C.2 *Throughout the algorithm, for every pair $v, v' \in V(H)$ of regular vertices, if $\mathsf{dist}_H(v, v') \leq D$, then $\mathsf{dist}_{\hat{H}}(v, v') \leq 20 \cdot \mathsf{dist}_H(v, v') + 8D'$.*

Proof: Consider some pair $v, v' \in V(H)$ of regular vertices, with $\mathsf{dist}_H(v, v') \leq D$. Let P be the shortest v-v' path in graph H, whose length $\ell(P) = \mathsf{dist}_H(v, v')$ is denoted by ℓ^*. We assume first that P contains at least one long edge, so $\ell^* \geq D'$.

Let $\left\{e^1, \ldots, e^k\right\}$ be the set of all long edges that appear on path P, indexed in the order of their appearance on P.

For every long edge e^i, we denote its endpoints by v_i and u_i, where u_i is a supernode. Let $\hat{e}^i$ be the unique edge on path P that shares the endpoint u_i with e^i. For all $1 \leq i \leq k$, we delete both e^i and

$\hat{e}^i$ from path P. Let $\mathcal{P} = \{P_1, \ldots, P_q\}$ denote the resulting set of subpaths of P (we do not include subpaths consisting of a single vertex), that are indexed in the order of their appearance on path P. Notice that, by deleting, for all $1 \leq i \leq k$, both edge e^i and $\hat{e}^i$, we have ensured that both endpoints of each path in $\mathcal{P}$ are regular vertices.

We say that a path $P_i \in \mathcal{P}$ is *long* iff its length (in graph H') is greater than $4D'$, and we say that it is short otherwise.

Consider first some short path $P_i \in \mathcal{P}$, and let $\hat{v}, \hat{v}'$ be its endpoints. Recall that $\mathcal{C}'$ is a set of clusters maintained as a solution to the $\mathsf{RecDynNC}$ on input structure $\mathcal{I}'$, with graph H' and distance bound $4D'$. Since $\mathsf{dist}_{H'}(v, v') \leq \ell_{H'}(P_i) \leq 4D'$, there is a cluster $C \in \mathcal{C}'$ containing both v and v'. Therefore, there is a vertex $u(C)$ in graph $\hat{H}$, and edges $(\hat{v}, u(C)), (\hat{v}', u(C))$ of length $4D'$ each. We let Q_i be the path obtained by concatenating these two edges. Then Q_i is a path in graph $\hat{H}$, connecting the endpoints of P_i, of length at most $8D'$.

Next, we consider a long path $P_i \in \mathcal{P}$, whose length must be greater than $4D'$. Let x_0 denote the first endpoint of P_i, and let y denote its last endpoint. Since the length of every edge in graph H' (and hence on path P_i) is at most D', we can find a collection $x_1, x_2, \ldots, x_r$ of **regular** vertices on path P_i, that appear on P_i in this order, such that, if we denote $x_{r+1} = y$, and, for all $0 \leq j \leq r$, we denote the subpath of P_i from x_j to x_{j+1} by R_j, then for all $0 \leq j < r$, the length of R_j is at least D' and at most $4D'$, and the length of R_r is at most $4D'$.

As before, since, for every regular vertex $v'' \in V(H')$, some cluster $C \in \mathcal{C}'$ must contain $B_{H'}(v'', 4D)$, we get that for all $0 \leq j \leq r$, there is a vertex $u(C_j)$ in graph $\hat{H}$, and edges $(x_j, u(C_j))$, $(u(C_j), x_{j+1})$ of length $4D'$ each. By concatenating all such edges, we obtain a path Q_i in graph $\hat{H}$, connecting x_0 to x_{r+1}, of length at most $8(r+1)D'$. From our definition of vertices $x_1, \ldots, x_r$, the length of path P_i in H' is at least rD'. Since the length of path P_i is greater than $4D'$, we are guaranteed that $r \geq 1$. Therefore, the length of path Q_i in graph $\hat{H}$ is at most $16\ell_{H'}(P_i)$.

Let $\mathcal{Q} = \{Q_i \mid P_i \in \mathcal{P}\}$. From the above discussion, the total length of all paths in $\mathcal{Q}$ (in graph $\hat{H}$) is at most $16\sum_i \ell_{H'}(P_i) + 8|\mathcal{P}|D' \leq 16\ell_H(P) \leq 16\ell^*$ (we have used the fact that $|\mathcal{P}| \leq k+1$, where k is the number of the long edges on path P, and each such long edge has length at least D').

Lastly, recall that for every long edge e^i, we have deleted e^i and $\hat{e}^i$ from path P. The length of edge e^i in graph $\hat{H}$ is at most $2\ell_{H'}(e^i)$, and the length of edge $\hat{e}^i$ in graph $\hat{H}$ is at most $\max\left\{D', 2\ell_{H'}(\hat{e}^i)\right\}$. Since the length of e^i in graph H' is at least D', the sum of lengths of both edges in graph $\hat{H}$ is bounded by $4\ell_{H'}(e^i) + 4\ell_{H'}(\hat{e}^i)$. Therefore, by concatenating all paths in $\mathcal{Q}$, and edges in $\left\{e^i, \hat{e}^i\right\}_{i=1}^k$, we obtain a path Q^* in graph $\hat{H}$, connecting v to v', whose length is at most $20\ell^*$.

Assume now that path P contains no long edge. Then we process path P exactly like we processed paths in $\mathcal{P}$: namely, if the length of P in graph H' is at most $4D'$, then we are guaranteed that there is some vertex $u(C)$, with edges $(v, u(C))$, $(v', u(C))$ of length $4D'$ each in graph $\hat{H}$, so $\mathsf{dist}_{\hat{H}}(v, v') \leq 8D'$. Otherwise, we treat P like a long path in $\mathcal{P}$, obtaining a path of length at most $16\ell_{H'}(P)$ connecting v to v' in graph $\hat{H}$. □

Next, we show that any path connecting a pair of regular vertices in graph $\hat{H}$ can be efficiently transformed into a path in graph H, connecting the same pair of vertices, without increasing the path length by much.

Claim C.3 *There is a deterministic algorithm, that we call* AlgTransformPath*, that, given a path P in graph $\hat{H}$, connecting a pair v, v' of regular vertices, computes a path P' in graph H, connecting the same pair of vertices, such that $\ell_H(P') \leq \alpha_{i-1} \cdot \hat{\ell}_{\hat{H}}(v, v')$. The running time of the algorithm is $O(|E(P')|)$.*

Proof: We process every supernode $u(C)$ on path P, with $C \in \mathcal{C}'$ one by one. Consider any such supernode, and let v_C, v'_C be the regular vertices of $\hat{H}$ appearing immediately before and immediately after $u(C)$ on path P. We perform query $\mathsf{short\text{-}path\text{-}query}(C, v_C, v'_C)$ to the data structure $\mathcal{D}(H')$, that maintains a solution to $\mathsf{RecDynNC}$ problem on graph H', and obtain a path P_C, of length at most $\alpha_{i-1} \cdot D'$, connecting v to v' in H, in time $O(|E(P_C)|)$. Notice that, since the lengths of the edges $(v_C, u(C)), (v'_C, u(C))$ are $4D'$ each, after this step is completed for every supernode $u(C)$ on path P with $C \in \mathcal{C}''$, we obtain a path P' in graph H, connecting v to v', whose length is at most $\alpha_{i-1} \cdot \hat{\ell}_{\hat{H}}(v, v')$. $\square$

Remainder of the Algorithm. Consider now the dynamic graph $\hat{H}$. Recall that the length of every edge in graph $\hat{H}$ is an integral multiple of D'.

Let $\hat{H}'$ be a graph that is identical to $\hat{H}$, except that for every edge $e \in E(\hat{H})$, we set its new length $\hat{\ell}'(e) = \hat{\ell}(e)/D'$ (recall that $\hat{\ell}(e)$ is the length of e in graph $\hat{H}$).

We set $\hat{D} = 50D/D'$. Since $D \leq 6W^{\epsilon i}$, while $D' = \lfloor W^{\epsilon(i-1)}/4 \rfloor$, we get that $\hat{D} = \Theta(W^{\epsilon})$.

We have now defined a valid input structure $\hat{\mathcal{I}}' = \left(\hat{H}', \left\{\hat{\ell}'(e)\right\}_{e \in E(\hat{H}')}, \hat{D}\right)$. Using the algorithm from Claim C.1, we obtain an online sequence $\hat{\Sigma} = (\hat{\Sigma}_1 \circ \hat{\Sigma}_2 \circ \cdots)$ of valid update operations for graph $\hat{H}'$, with dynamic degree bound $\mu \leq W^{O(1/\log\log W)}$; recall that initially, the number of regular vertices in $\hat{H}$ is at most W. We use the algorithm from Theorem 3.3 in order to maintain a solution to the $\mathsf{RecDynNC}$ problem on graph $\hat{H}'$, with distance bound $\hat{D}$, and approximation factor:

$$\hat{\alpha} = (\log(W\mu))^{2^{O(1/\epsilon)}} \leq (\log W)^{2^{O(1/\epsilon)}} \leq (\log W)^{2^{\tilde{c}/\epsilon}},$$

since $\mu \leq W^{O(1/\log\log W)}$, and $\tilde{c}$ is a large constant.

Recall that the total update time of this algorithm is:

$$O\left(W^{1+O(\epsilon)} \cdot \mu^{2+O(\epsilon)} \cdot \hat{D}^3 \cdot (\log(W\mu))^{O(1/\epsilon^2)}\right) \leq O\left(W^{1+O(\epsilon)} \cdot (\log W)^{O(1/\epsilon^2)}\right),$$

since $\hat{D} = O(W^{\epsilon})$, $\mu \leq W^{O(1/\log\log W)}$, and $\epsilon \geq c/\log\log W$. We denote the corresponding data structure by $\mathcal{D}(\hat{H}')$. The corresponding neighborhood cover is denoted by $\hat{\mathcal{C}}$, and we denote by $\hat{\mathcal{U}} = \left\{V(C) \mid C \in \hat{\mathcal{C}}\right\}$. We also denote the algorithm that we have just described, for maintaining the data structure $\mathcal{D}(\hat{H}')$, by Alg_2.

We are now ready to complete the description of our algorithm for solving the $\mathsf{RecDynNC}$ problem on graph H. Recall that $V(H) \subseteq V(\hat{H}')$. The neighborhood cover $\mathcal{C}$ that we maintain is defined as follows: for every vertex set $S \in \hat{\mathcal{U}}$, we let $S' = S \cap V(H)$. We then define $\mathcal{U} = \left\{S' \mid S \in \hat{\mathcal{C}}'\right\}$, and we let $\mathcal{C}$ contain, for each vertex set $S' \in \mathcal{U}$ the graph $H[S']$. Since we are guaranteed that the collection $\hat{\mathcal{U}}$ of vertex subsets of $\hat{H}'$ only undergoes allowed changes, it is easy to see that so does $\mathcal{U}$.

Consider any regular vertex $v \in V(H)$ at any point in the algorithm's execution, and vertex set $X = B_H(v, D)$. Recall that, from Lemma C.2, for every regular vertex $v' \in X$, $\mathsf{dist}_{\hat{H}}(v, v') \leq 20 \cdot \mathsf{dist}_H(v, v') + 8D'$, and so $\mathsf{dist}_{\hat{H}'}(v, v') \leq (20\mathsf{dist}_H(v, v') + 8D')/D' \leq \hat{D}$. Similarly, if $u \in X$ is a supernode, then there is a regular vertex $v' \in X$, that is a neighbor of u, with $\mathsf{dist}_H(v, v') \leq \mathsf{dist}_H(v, u) - \ell_H(v', u)$. We then get that $\mathsf{dist}_{\hat{H}}(v, v') \leq 20 \cdot \mathsf{dist}_H(v, v') + 8D'$, and $\mathsf{dist}_{\hat{H}}(v, u) \leq 20 \cdot \mathsf{dist}_H(v, v') + 9D' + \ell_H(v', u) \leq 20D + 9D'$. Therefore, $\mathsf{dist}_{\hat{H}'}(v, u) \leq (20D + 9D')/D' \leq \hat{D}$. We conclude that $X = B_H(v, D) \subseteq B_{\hat{H}'}(v, \hat{D})$. Let $\hat{C} = \mathsf{CoveringCluster}(v)$ be the cluster of $\hat{\mathcal{C}}$ containing

$B_{\hat{H}'}(v, \hat{D}')$. Denote $S = V(\hat{C})$, and let $S' = S \cap V(H)$ be the corresponding vertex set in $\mathcal{U}$. Then $B_H(v, D) \subseteq S'$. Therefore, if we denote by C the cluster of $\mathcal{C}$ with $V(C) = S'$, then we can set $\mathsf{CoveringCluster}(v)$ in $\mathcal{C}$ to be C.

Using the data structure $\mathcal{D}(\hat{H}')$, it is then immediate to maintain, for every regular vertex $v \in V(H)$, a cluster $C = \mathsf{CoveringCluster}(v)$ in $\mathcal{C}$, with $B_H(v, D) \subseteq V(C)$. We can also maintain, for every vertex $x \in V(H)$, a list $\mathsf{ClusterList}(x) \subseteq \mathcal{C}$ of clusters containing x, and for every edge $e \in E(H)$, a list $\mathsf{ClusterList}(e) \subseteq \mathcal{C}$ of clusters containing e, using similar data structures for graph $\hat{H}'$.

Recall that the algorithm from Theorem 3.3, that we used in order to maintain neighborhood cover $\hat{\mathcal{C}}'$ in graph $\hat{H}'$ ensures that for every regular vertex $v \in V(\hat{H}')$, the total number of clusters in the neighborhood cover $\hat{\mathcal{C}}$ that the algorithm maintains, to which vertex v ever belonged is bounded by $W^{O(1/\log\log W)}$. Therefore, for every regular vertex $v \in V(H)$, the total number of clusters in the cluster set $\mathcal{C}$ to which v may ever belong is also bounded by $W^{O(1/\log\log W)}$.

Lastly, we show an algorithm for responding to queries $\mathsf{short\text{-}path\text{-}query}(C, v, v')$, where C is a cluster in $\mathcal{C}$, and $v, v' \in V(C)$ are regular vertices lying in C. Denote $S' = V(C)$, and let $\hat{C} \in \hat{\mathcal{C}}$ be the corresponding cluster (with $S' \subseteq V(\hat{C})$). We run query $\mathsf{short\text{-}path\text{-}query}(\hat{C}, v, v')$ in data structure $\mathcal{D}(\hat{H}')$, obtaining a path P in graph $\hat{H}'$, connecting v to v', of length at most $\hat{D} \cdot \hat{\alpha} \leq (50D \cdot (\log W)^{2^{\tilde{c}/\epsilon}})/D'$ (we have used the fact that $\hat{D} = 50D/D'$ and $\hat{\alpha} = (\log W)^{2^{\tilde{c}/\epsilon}}$). Note that the length of path P in graph $\hat{H}$ is at most $50D \cdot (\log W)^{2^{\tilde{c}/\epsilon}} \leq D \cdot (\log W)^{2 \cdot 2^{\tilde{c}/\epsilon}}$. The time required to process query $\mathsf{short\text{-}path\text{-}query}(\hat{C}, v, v')$ in data structure $\mathcal{D}(\hat{H}')$ is $O(|E(P)|)$. Lasly, we apply Algorithm AlgTransformPath from Claim C.3 to path P in graph $\hat{H}$, to obtain a path P' in graph H, connecting v to v', whose length is bounded by:

$$\begin{aligned}
\alpha_{i-1} \cdot \hat{\ell}_{\hat{H}}(v, v') &\leq \alpha_{i-1} \cdot D \cdot (\log W)^{2 \cdot 2^{\tilde{c}/\epsilon}} \\
&\leq D \cdot (\log W)^{\tilde{c}(i-1) \cdot 2^{\tilde{c}/\epsilon}} \cdot (\log W)^{2 \cdot 2^{\tilde{c}/\epsilon}} \\
&\leq D \cdot (\log W)^{\tilde{c} i \cdot 2^{\tilde{c}/\epsilon}} \\
&= D \cdot \alpha_i.
\end{aligned}$$

The running time of the algorithm is $O(|E(P')|)$.

From the above discussion, and the fact that our algorithm supports $\mathsf{short\text{-}path\text{-}query}$ queries, it is immediate to verify that, throughout the algorithm, $\mathcal{C}$ is a weak $(D, \alpha_i \cdot D)$-neighborhood cover of the regular vertices of H.

Total Update Time. We now bound the total update time of the algorithm. The update time is dominated by the update time of the algorithm from Claim C.1, and the algorithm Alg_2. The former has update time $O\left(\tilde{c}^{i-1} \cdot W^{1+\tilde{c}\epsilon} \cdot (\log W)^{\tilde{c}/\epsilon^2}\right)$, while the latter has update time $O\left(W^{1+O(\epsilon)} \cdot (\log W)^{O(1/\epsilon^2)}\right)$. Since we can assume that $\tilde{c}$ is a sufficiently large constant, the total update time of the algorithm is bounded by $\left(\tilde{c}^{i} \cdot W^{1+\tilde{c}\epsilon} \cdot (\log W)^{\tilde{c}/\epsilon^2}\right)$.

D Proofs Omitted from Section 4

D.1 Proof of Claim 4.2

Assume otherwise. Then for all $1 < i < 128 \log^4 W$, layer L_i is ineligible. Since the total weight of all vertices in $B_C\left(x, 256D\log^4 W\right)$ is at most $W(C)/2$, Condition C1 holds for all $1 < i < \lfloor 128 \log^4 W \rfloor$. For an index $1 < i < \lfloor 128 \log^4 W \rfloor$, we say that layer L_i is *type-0 ineligible* iff Condition C2 is violated for it, and we say that it is *type-j ineligible*, for some $1 \le j \le r$, if Condition C3 for the index j is violated for it. The following two observations, whose proofs are standard, bound the number of ineligible layers of each of these types.

Observation D.1 *The total number of type-0 ineligible layers is at most* $128 \log^3 W$.

Proof: Assume otherwise. Denote $y = \lceil 128 \log^3 W \rceil - 2$. Let $1 < i_1 < i_2 < \cdots < i_y$ be indices of y type-0 ineligible layers. From the definition of a type-1 ineligible layer, for all $1 \le q \le y$:

$$W_{i_q} > \frac{\sum_{i'<i_q} W_{i'}}{64 \log^2 W} \ge \frac{\sum_{q'<q} W_{i_{q'}}}{64 \log^2 W}.$$

Therefore, if we denote, for all $1 \le q \le y$, $A_q = \sum_{q'<q} W_{i_{q'}}$, then for all $1 < q \le y$, $A_q \ge \left(1 + \frac{1}{64\log^2 W}\right) A_{q-1}$ must hold. Since the weight of every regular vertex is at least 1, and each edge of H has length at most D, $W_1 \ge 1$ holds, and therefore:

$$A_y \ge \left(1 + \frac{1}{64 \log^2 W}\right)^y > W,$$

since $y = \lceil 128 \log^3 W \rceil - 2$. This is a contradiction, since W is the total weight of all vertices of H at the beginning of the algorithm, and, as the algorithm progresses, this weight may only decrease. □

Observation D.2 *For all $1 \le j \le r$, the number of type-j ineligible layers is at most* $128 \log^3 W$.

Proof: Assume that the observation is false for some $1 \le j \le r$. Denote $z = \lceil 128 \log^3 W \rceil - 2$, and let $1 < i_1 < i_2 < \cdots < i_z$ be indices of z type-j ineligible layers. From the definition of a type-j ineligible layer, for all $1 \le q < z'$:

$$W(S_{\ge j} \cap L_{i_q}) > \frac{\sum_{i'<i_q} W(L_{i'} \cap S_{\ge j})}{64 \log^2 W} \ge \frac{\sum_{q'=1}^{q-1} W(L_{i_{q'}} \cap S_{\ge j})}{64 \log^2 W}.$$

Therefore, if we denote, for all $1 \le q \le y$, $A_q = \sum_{q'<q} W(L_{i_{q'}} \cap S_{\ge j})$, then for all $1 < q \le z$, $A_q \ge \left(1 + \frac{1}{64\log^2 W}\right) A_{q-1}$ must hold. Moreover, since i_1 is a type-j ineligible layer, $W(S_{\ge j} \cap L_{i_1}) \ge 1$ must hold. Therefore:

$$A_z \ge \left(1 + \frac{1}{64 \log^2 W}\right)^{z-1} > W,$$

since $z = \lceil 128 \log^3 W \rceil - 2$, a contradiction. □

We have shown that for all $0 \le j \le r$, the total number of type-j ineligible layers is bounded by $128 \log^3 W$. Since $r < \log W - 2$, we conclude that the total number of ineligible layers L_i, for $i > 1$, is at most $128 \log^4 W - 2$, a contradiction.

D.2 Proof of Claim 4.4

Consider some vertex $y \in V(C)$. Recall that $V(C') = L_1 \cup \cdots \cup L_i = B_C(x, 2Di)$. We now consider two cases. First, if $y \in B_C(x, 2Di - D)$, then clearly $B_C(y, D) \subseteq C'$. Otherwise, $y \notin B_C(x, 2Di - D)$, and so $B_C(y, D) \cap B_C(x, 2Di - 2D) = \emptyset$. Since cluster C'' contains all vertices of C except those lying in $B_C(x, 2Di - 2D) = L_1 \cup \cdots \cup L_{i-1}$, we get that $B_C(y, D) \subseteq V(C'')$.

D.3 Proof of Lemma 4.5

We start by proving that $\beta' \leq \beta$. We assume that ProcCut chose a layer L_i of C as an eligible layer, and we denote by C' and C'' the resulting two clusters. Recall that for every cluster $\hat{C}$ and for every vertex $y \in \hat{C}$, the budget of y with respect to $\hat{C}$ is $\beta_{\hat{C}}(y) = \left(1 + \frac{\log W(\hat{C})}{\log^2 W}\right) \cdot w(y)$. Therefore, it is easy to verify that for every vertex $y \in V(C'') \setminus L_i$, $\beta_{C''}(y) \leq \beta_C(y)$. We conclude that:

$$\sum_{y \in V(C'') \setminus L_i} \beta_{C''}(y) \leq \sum_{y \in V(C'') \setminus L_i} \beta_C(y). \tag{2}$$

Consider now some vertex $y \in V(C') \setminus L_i$. Since, by Condition C1, $W(C') \leq W(C)/2$, we get that $\log(W(C')) \leq \log(W(C)) - 1$. Therefore:

$$\beta_{C'}(y) = \left(1 + \frac{\log(W(C'))}{\log^2 W}\right) \cdot w(y) \leq \left(1 + \frac{\log(W(C))}{\log^2 W}\right) \cdot w(y) - \frac{w(y)}{\log^2 W}.$$

We conclude that:

$$\sum_{y \in V(C') \setminus L_i} \beta_{C'}(y) \leq \sum_{y \in V(C') \setminus L_i} \beta_C(y) - \frac{W(V(C') \setminus L_i)}{\log^2 W}. \tag{3}$$

Consider now some vertex $y \in L_i$. Before ProcCut was executed, the copy of vertex y in cluster C contributed $\beta_C(y)$ to the total budget β. At the end of the procedure, we have created two copies of y, one of which lies in C' and another in C''. Note that $\beta_{C''}(y) \leq \beta_C(y)$, while $\beta_{C'}(y) \leq 2w(y)$. Therefore, the contribution of these two copies of y to the total budget β' is bounded by:

$$\beta_{C'}(y) + \beta_{C''}(y) \leq 2w(y) + \beta_C(y).$$

The total contribution of the vertices of L_i to the new budget β' is then bounded by:

$$2W(L_i) + \sum_{y \in L_i} \beta_C(y).$$

From Condition C2, $W(L_i) \leq W(L_1 \cup \cdots \cup L_{i-1})/(64 \log^2 W) = W(V(C') \setminus L_i)/(64 \log^2 W)$. Therefore, we get that:

$$\sum_{y \in L_i} (\beta_{C'}(y) + \beta_{C''}(y)) \leq \frac{W(V(C') \setminus L_i)}{32 \log^2 W} + \sum_{y \in L_i} \beta_C(y). \tag{4}$$

Lastly, by summing up bounds from Equations 2, 3 and 4, we get that:

$$\sum_{y\in V(C')}\beta_{C'}(y)+\sum_{y\in V(C'')}\beta_{C''}(y)\leq \sum_{y\in V(C)}\beta_C(y)-\frac{W(V(C')\setminus L_i)}{\log^2 W}+\frac{W(V(C')\setminus L_i)}{32\log^2 W}\leq \sum_{y\in V(C)}\beta_C(y).$$

Note that the only values $\beta_{\hat{C}}(y)$ that change are for cluster $\hat{C}=C$ and vertices $y\in V(C)$. Therefore, we conclude that $\beta'\leq\beta$.

Next, we fix an index $1\leq j\leq r$, and we prove that $\beta'_{\geq j}\leq \beta_{\geq j}+2^j(1+1/\log W)W'_j$. In order to do so, we think of the procedure ProcCut as consisting of two steps. In the first step, we compute the two clusters C' and C'' as before, but we do not update yet the classes of the vertices. That is, if, for some vertex y, a new copy of y was introduced due to the procedure, and y needs to move from class S_a to class S_{a+1}, we do not move vertex y yet. The total budget $\beta'_{\geq j}$ of the vertices lying in classes $S_{\geq j}$ at the end of this step is calculated with respect to the original classes. We will show that at the end of the first step, $\beta'_{\geq j}\leq\beta_{\geq j}$ holds. Then in the second step we appropriately update the class of each vertex. If $\hat{S}_j$ is the set of all vertices that are added to class S_j during the second step, then each such vertex $y\in\hat{S}_j$ has $n_y=2^j$, and, for every cluster $\hat{C}$ containing y, $\beta_{\hat{C}}(y)=\left(1+\frac{\log W(\hat{C})}{\log^2 W}\right)\cdot w(y)\leq\left(1+\frac{1}{\log W}\right)\cdot w(y)$. Therefore, the increase in $\beta'_{\geq j}$ due to the second step is bounded by $2^j\cdot(1+1/\log W)\cdot W(\hat{S}_j)=2^j\cdot(1+1/\log W)W'_j$. In order to complete the proof of the lemma, it is enough to show that, at the end of the first step, $\beta'_{\geq j}\leq\beta_{\geq j'}$ holds. The proof is very similar to the proof of the first part of the lemma. Recall that the layer L_i that was chosen by the procedure is an eligible layer. Therefore, $W(S_{\geq j}\cap L_i)\leq\left(\sum_{i'<i}W(L_{i'}\cap S_{\geq j})\right)/\left(64\log^2 W\right)$ must hold.

As before, for every vertex $y\in V(C'')\setminus L_i$, $\beta_{C''}(y)\leq\beta_C(y)$ must hold, and so:

$$\sum_{y\in(V(C'')\cap S_{\geq j})\setminus L_i}\beta_{C''}(y)\leq\sum_{y\in(V(C'')\cap S_{\geq j})\setminus L_i}\beta_C(y). \tag{5}$$

Consider now some vertex $y\in V(C')\setminus L_i$. Since, by Condition C1, $W(C')\leq W(C)/2$, we get that $\log(W(C'))\leq\log(W(C))-1$. Therefore, as before:

$$\beta_{C'}(y)\leq\left(1+\frac{\log W(C')}{\log^2 W}\right)\cdot w(y)\leq\left(1+\frac{\log W(C)}{\log^2 W}\right)\cdot w(y)-\frac{w(y)}{\log^2 W}.$$

We conclude that:

$$\sum_{y\in(V(C')\cap S_{\geq j})\setminus L_i}\beta_{C'}(y)\leq\sum_{y\in(V(C')\cap S_{\geq j})\setminus L_i}\beta_C(y)-\frac{W(V(C'\cap S_{\geq j})\setminus L_i)}{\log^2 W}. \tag{6}$$

Consider now some vertex $y\in L_i\cap S_{\geq j}$. Before ProcCut was executed, the copy of vertex y in cluster C contributed $\beta_C(y)$ to the budget $\beta_{\geq j}$. At the end of the procedure, we have created two copies of y, one of which lies in C' and another in C''. Exactly as before, the contribution of these two copies of y to the total budget β' is bounded by:

$$\beta_{C'}(y)+\beta_{C''}(y)\leq 2w(y)+\beta_C(y).$$

Therefore, the total contribution of the vertices of $L_i\cap S_{\geq j}$ to the new budget $\beta'_{\geq j}$ is bounded by:

$$2W(L_i \cap S_{\geq j}) + \sum_{y \in L_i \cap S_{\geq j}} \beta_C(y).$$

From Condition C3, $W(L_i \cap S_{\geq j}) \leq \left(\sum_{i'<i} W(L_{i'} \cap S_{\geq j})\right) / \left(64 \log^2 W\right) = W((C' \cap S_{\geq j}) \setminus L_i) / \left(64 \log^2 W\right)$. Therefore, we get that:

$$\sum_{y \in L_i \cap S_{\geq j}} (\beta_{C'}(y) + \beta_{C''}(y)) \leq W((C' \cap S_{\geq j}) \setminus L_i) / \left(32 \log^2 W\right) + \sum_{y \in L_i \cap S_{\geq j}} \beta_C(y). \tag{7}$$

Lastly, by summing up bounds from Equations 5, 6 and 7, we get that:

$$\begin{aligned}\sum_{y \in V(C') \cap S_{\geq j}} \beta_{C'}(y) + &\sum_{y \in V(C'') \cap S_{\geq j}} \beta_{C''}(y) \\ &\leq \sum_{y \in V(C) \cap S_{\geq j}} \beta_C(y) - \frac{W((V(C') \cap S_{\geq j}) \setminus L_i)}{\log^2 W} + \frac{W((V(C') \cap S_{\geq j}) \setminus L_i)}{32 \log^2 W} \\ &\leq \sum_{y \in V(C) \cap S_{\geq j}} \beta_C(y).\end{aligned}$$

We conclude that, at the end of the first step, $\beta'_{\geq j} \leq \beta_{\geq j}$ holds.

E Proofs Omitted from Section 5

E.1 Proof of Theorem 5.2

We start with the following lemma, whose proof uses standard techniques and is deferred to the following subsection.

Lemma E.1 *There is a constant $c \geq 1$ and a deterministic algorithm, that, given a graph G with integral lengths $\ell(e) \geq 1$ on its edges $e \in E(G)$, a subset S of its vertices, where $|S| = k$ is an even integer, and parameters $\eta, D' \geq 1$ and $\epsilon \geq \Omega(1/\log k)$, computes one of the following:*

- *either a graph X with $V(X) \subseteq S$, and $|V(X)| \geq k/2$, such that the maximum vertex degree in X is at most $O(\log k)$, and X is a φ-expander, for $\varphi = 1/(\log k)^{c/\epsilon}$, together with an embedding $\mathcal{P}$ of X into G, with congestion at most η, such that every path in $\mathcal{P}$ has length at most D'; or*
- *a subset E' of at most $\frac{ckD' \log^2 k}{\eta}$ edges of G, and two disjoint subsets $S_1, S_2 \subseteq S$ of vertices of cardinality at least $k/(\log k)^{c/\epsilon}$ each, such that, in graph $G \setminus E'$, $\mathsf{dist}(S_1, S_2) > D'$.*

The running time of the algorithm is $\widetilde{O}\left(k^{1+\epsilon} (\log k)^{O(1/\epsilon^2)}\right) + \widetilde{O}\left(|E(G)| D' \eta\right)$.

We prove Lemma E.1 in Appendix E.1.1, after we complete the proof of Theorem 5.2 using it. We start with the following corollary of Lemma E.1.

Corollary E.2 *There is a constant $c' \geq 1$, and a deterministic algorithm, that, given a graph G with non-negative lengths $\ell(e)$ on its edges $e \in E(G)$, a subset S of its vertices, where $|S| \leq k$ for some parameter k, and $|S|$ is an even integer, together with parameters $\eta, D'' \geq 1$ and $\epsilon \geq \Omega(1/\log k)$, computes one of the following:*

- *either a graph X with $V(X) \subseteq S$, and $|V(X)| \geq |S|/2$, such that the maximum vertex degree in X is at most $O(\log k)$, and X is a φ-expander, for $\varphi = 1/(\log k)^{c'/\epsilon}$, together with an embedding $\mathcal{P}$ of X into G, with congestion at most η, such that every path in $\mathcal{P}$ has length at most $64D''\log^2 k$; or*
- *a subset E' of at most $\frac{c'|S|D''\log^4 k}{\eta}$ edges of G, and a partition (S'_0, S'_1, S'_2) of S into disjoint subsets, such that $|S'_1|, |S'_2| \geq |S|/(\log k)^{c'/\epsilon}$, $|S'_0| \leq \frac{\min\{|S'_1|,|S'_2|\}}{8\log k}$, and, in graph $G \setminus E'$, $\mathsf{dist}(S'_1, S'_2) > D''$.*

The running time of the algorithm is $\widetilde{O}\left(k^{1+\epsilon}(\log k)^{O(1/\epsilon^2)}\right) + \widetilde{O}\left(|E(G)|D''\eta\right)$.

Proof: We assume that c' is a large enough constant, and in particular $c' > c$, where c is the constant from the statement of Lemma E.1. We set $D' = 64D''\log^2 k$, and apply the algorithm from Lemma E.1 to graph G, with the new parameter D', and parameters η, ϵ that remain unchanged. We now consider two cases. In the first case, the outcome of the algorithm from Lemma E.1 is a graph X with $V(X) \subseteq S$, and $|V(X)| \geq |S|/2$, such that the maximum vertex degree in X is at most $O(\log|S|) \leq O(\log k)$, and X is a φ-expander, for $\varphi = 1/(\log|S|)^{c/\epsilon} \geq 1/(\log k)^{c'/\epsilon}$, together with an embedding $\mathcal{P}$ of X into G, with congestion at most η, such that every path in $\mathcal{P}$ has length at most $D' = 64D''\log^2 k$. In this case, we terminate the algorithm, and return the graph X and its embedding $\mathcal{P}$ as the algorithm's outcome.

From now on we assume that the second case happened, and the algorithm from Lemma E.1 returned a subset E' of at most $\frac{c|S|D'\log^2 k}{\eta} \leq \frac{c'|S|D''\log^4 k}{\eta}$ edges of G, and two disjoint subsets $S_1, S_2 \subseteq S$ of vertices of cardinality at least $|S|/(\log|S|)^{c/\epsilon} \geq |S|/(\log k)^{c'/\epsilon}$ each, such that, in graph $G \setminus E'$, $\mathsf{dist}(S_1, S_2) > D'$.

In the remainder of the algorithm, we will compute the desired partition (S'_0, S'_1, S'_2) by using standard ball-growing technique (somewhat similar to the one employed in Procedure $\mathsf{ProcCut}$ from Section 4.2).

We let $L^1_0 = S_1$. For $i \geq 1$, we define the ith layer for S_1 as $L^1_i = B_{G\setminus E'}(S_1, 2iD'') \setminus B_{G\setminus E'}(S_1, 2(i-1)D'')$. We denote by $K^1_i = |S \cap L^1_i|$, and by $J^1_i = K^1_0 + \cdots + K^1_i$, where $K^1_0 = |S_1|$.

For $i \geq 1$, we say that layer L^1_i is *eligible* iff:

- $K^1_i \leq J^1_{i-1}/(8\log k)$; and
- $J^1_i \leq |S|/2$.

Similarly, we let $L^2_0 = S_2$, and, for $i \geq 1$, we define the ith layer for S_2 as $L^2_i = B_{G\setminus E'}(S_2, 2iD'') \setminus B_{G\setminus E'}(S_2, 2(i-1)D'')$. We denote by $K^2_i = |S \cap L^2_i|$, and by $J^2_i = K^2_0 + \cdots + K^2_i$, where $K^2_0 = |S_2|$. As before, for $i \geq 1$, we say that layer L^2_i is *eligible* iff:

- $K^2_i \leq J^2_{i-1}/(8\log k)$; and
- $J^2_i \leq |S|/2$.

We need the following simple observation.

Observation E.3 *There is an integer $1 \le i \le 8\log^2 k$, such that either L^1_i or L^2_i is an eligible layer.*

Proof: Recall that $\mathsf{dist}_{G\setminus E'}(S_1,S_2) > D'$, and, since $D' = 64D''\log^2 k$, the vertex sets $B_{G\setminus E'}(S_1, 32D''\log^2 k)$ and $B_{G\setminus E'}(S_2, 32D''\log^2 k)$ are disjoint. Let $i' = \lfloor 16\log^2 k \rfloor$. Then either $J^1_{i'} \le |S|/2$ or $J^2_{i'} \le |S|/2$ must hold. Assume without loss of generality that it is the former. We now prove that there must be some integer $1 \le i \le i'$, such that layer L^1_i is eligible.

Assume otherwise. Note that for all $1 \le i \le i'$, $J^1_i \le |S|/2$ must hold. Since we have assumed that layer L^1_i is ineligible, $K^1_i > J^1_{i-1}/(8\log k)$ must hold, and so $J^1_i \ge J^1_{i-1}(1 + 1/(8\log k))$. But then we get that:

$$J^1_{i'} \ge |S_1| \cdot \left(1 + \frac{1}{8\log k}\right)^{i'} \ge |S_1| \cdot \left(1 + \frac{1}{8\log k}\right)^{\lfloor 16\log^2 k\rfloor} \ge k,$$

a contradiction. □

We run two BFS searches in graph $G \setminus E'$ in parallel, one starting from S_1, and another starting from S_2, so that at any time point, both searches discover the same number of edges, until we encounter the first index $1 \le i \le 8\log^2 k$, such that either L^1_i or L^2_i is an eligible layer. Assume without loss of generality that L^1_i is an eligible layer. We then let S'_0 contain all vertices of $S \cap L^1_i$, S'_1 contain all vertices of S lying in $B_{G\setminus E'}(S_1, 2(i-1)D'')$, and S'_2 contain all remaining vertices of S. Clearly, $|S'_1| \le |S'_2|$, and, from the definition of an eligible layer, $|S'_0| \le |S'_1|/(8\log k)$. Moreover, $S_1 \subseteq S'_1$, so $|S'_1| \ge |S|/(\log k)^{c'/\epsilon}$, as required. Since $S'_1 \subseteq B_{G\setminus E'}(S_1, 2(i-1)D'')$, while $S'_2 \cap B_{G\setminus E'}(S_1, 2iD'') = \emptyset$, we get that $\mathsf{dist}_{G\setminus E'}(S'_1, S'_2) > D''$. It now remains to analyze the running time of the algorithm. The running time for the algorithm from Lemma E.1 is $\widetilde{O}\left(|S|^{1+\epsilon}(\log k)^{O(1/\epsilon^2)}\right) + \widetilde{O}\left(|E(G)|D'\eta\right) = \widetilde{O}\left(|S|^{1+\epsilon}(\log k)^{O(1/\epsilon^2)}\right) + \widetilde{O}\left(|E(G)|D''\eta\right)$. The additional running time required to compute an eligible layer and the vertex sets S'_0, S'_1, S'_2 is bounded by $O(|E(G)|)$. therefore, the total running time of the algorithm is $\widetilde{O}\left(|S|^{1+\epsilon}(\log k)^{O(1/\epsilon^2)}\right) + \widetilde{O}\left(|E(G)|D''\eta\right)$. □

We need the following, somewhat stronger corollary, that allows us to either compute an expander X and its embedding into G as before, or to compute a number of subsets of the set S that are far from each other in $G \setminus E'$, but now we additionally guarantee that the cardinality of each subset is significantly smaller than the cardinality of the initial set S; as before, we also compute a small subset S_0 of vertices of S that we will discard.

Corollary E.4 *There is a constant $c'' \ge 1$, and a deterministic algorithm, that, given a graph G with non-negative lengths $\ell(e)$ on its edges $e \in E(G)$, a subset S of its vertices, where $|S| \le k$ for some parameter k, together with parameters $\eta, D'' \ge 1$ and $\epsilon \ge \Omega(1/\log k)$, such that $|S| \ge 40(\log k)^{c''/\epsilon}$ holds, computes one of the following:*

- *either a graph X with $V(X) \subseteq S$, and $|V(X)| \ge |S|/4$, such that the maximum vertex degree in X is at most $O(\log k)$, and X is a φ-expander, for $\varphi = 1/(\log k)^{c''/\epsilon}$, together with an embedding $\mathcal{P}$ of X into G, with congestion at most η, such that every path in $\mathcal{P}$ has length at most $64D''\log^2 k$; or*
- *a subset E' of at most $\frac{|S|D''(\log k)^{c''/\epsilon}}{\eta}$ edges of G, and a partition $(S''_0, S''_1, S''_2, \ldots, S''_q)$ of S into disjoint subsets, such that, for all $1 \le i \le q$, $|S|/(\log k)^{c''/\epsilon} \le |S''_i| \le |S|/2$, $|S''_0| \le |S|/(4\log k)$, and, in graph $G \setminus E'$, for all $1 \le i < i' \le q$, $\mathsf{dist}(S''_i, S''_{i'}) > D''$.*

The running time of the algorithm is $\widetilde{O}\left(\left(k^{1+\epsilon}(\log k)^{O(1/\epsilon^2)}+|E(G)|D''\eta\right)\cdot(\log k)^{O(1/\epsilon)}\right)$.

Proof: We assume that the constant c'' is large enough, so that $c'' \geq c'$, where c' is the constant from Corollary E.2. The algorithm consists of at most $(\log k)^{c''/\epsilon}$ iterations. Over the course of the algorithm, we maintain a subset S''_0 of S, another subset S^* of $S \setminus S''_0$, and a partition $\mathcal{S}$ of $S \setminus (S''_0 \cup S^*)$. We also maintain a collection $E' \subseteq E(G)$ of edges. We ensure that the following invariants hold throughout the algorithm:

- $|S''_0| \leq 1 + \sum_{S' \in \mathcal{S}} |S'|/(4\log k)$;
- $|S^*| > |S|/2$, and $|S^*|$ is even;
- For all $S', S'' \in \mathcal{S}$ with $S' \neq S''$, $\mathsf{dist}_{G\setminus E'}(S', S'') > D''$;
- For all $S' \in \mathcal{S}$, $\mathsf{dist}_{G\setminus E'}(S', S^*) > D''$; and
- For all $S' \in \mathcal{S}$, $|S|/(\log k)^{c''/\epsilon} \leq |S'| \leq |S|/2$.

At the beginning of the algorithm, if $|S|$ is even, then we set $S''_0 = \emptyset$, and $S^* = S$; otherwise, we let S''_0 contain an arbitrary vertex s of S, and we set $S^* = S \setminus \{s\}$. We also set $\mathcal{S} = \emptyset$, and $E' = \emptyset$. It is easy to verify that all invariants hold.

We now describe an execution of an iteration. We apply the algorithm from Corollary E.2 to graph $G \setminus E'$, vertex set S^*, and parameters k, η, D'' and ϵ. We now consider two cases.

In the first case, the algorithm returns a graph X with $V(X) \subseteq S^*$, and $|V(X)| \geq |S^*|/2 \geq |S|/4$, such that the maximum vertex degree in X is at most $O(\log k)$, and X is a φ-expander, for $\varphi = 1/(\log k)^{c'/\epsilon} \geq 1/(\log k)^{c''/\epsilon}$, together with an embedding $\mathcal{P}$ of X into G, with congestion at most η, such that every path in $\mathcal{P}$ has length at most $64D''\log^2 k$. We then terminate the algorithm, and return the expander X and its embedding.

In the second case, the algorithm from Corollary E.2 returns a subset E^* of at most $\frac{c'|S|D''\log^4 k}{\eta}$ edges of G, and a partition (S'_0, S'_1, S'_2) of S^* into disjoint subsets. We assume without loss of generality that $|S'_1| \leq |S'_2|$, so in particular $|S'_1| \leq |S|/2$. Recall that the algorithm guarantees that $|S'_1| \geq |S^*|/(\log k)^{c'/\epsilon} \geq |S|/(\log k)^{c''/\epsilon}$, and that $|S'_0| \leq |S'_1|/(8\log k)$. Additionally, in graph $G \setminus (E' \cup E^*)$, $\mathsf{dist}(S'_1, S'_2) > D''$. If $|S'_2|$ is odd, then we let $s' \in S'_2$ be any vertex, and we move s' from S'_2 to S'_0. Notice that, since $|S^*| \geq |S|/2 \geq 20(\log k)^{c''/\epsilon}$, while $|S'_1| \geq |S^*|/(\log k)^{c'/\epsilon}$, $|S'_0| \leq |S'_1|/(4\log k)$ still holds.

We add the edges of E^* to E', the vertices of S'_0 to S''_0, and the set S'_1 to $\mathcal{S}$. We also set $S^* = S'_2$. Since $|S''_0| \leq 1 + \sum_{S' \in \mathcal{S}} |S'|/(4\log k)$ held at the beginning of the current iteration, and $|S'_0| \leq |S'_1|/(4\log k)$, it is immediate to verify that $|S''_0| \leq 1 + \sum_{S' \in \mathcal{S}} |S'|/(4\log k)$ continues to hold.

It is also easy to verify that for all $S', S'' \in \mathcal{S}$ with $S' \neq S''$, $\mathsf{dist}_{G\setminus E'}(S', S'') > D''$ continues to hold. Indeed, if $S', S'' \neq S'_1$, then $\mathsf{dist}_{G\setminus E'}(S', S'') \geq D''$ held at the beginning of the current iterations, and, since we have only added edges to E', the inequality continues to hold at the end of the iteration. Otherwise, if, for example, $S'' = S'_1$, then, since $\mathsf{dist}_{G\setminus E'}(S', S^*) \geq D''$ held at the beginning of the current iteration, and $S'' \subseteq S^*$, $\mathsf{dist}_{G\setminus E'}(S', S'') \geq D''$ holds at the end of the current iteration. Using similar reasonings, it is easy to verify that, for all $S' \in \mathcal{S}$, $\mathsf{dist}_{G\setminus E'}(S', S^*) \geq D''$ holds at the end of the current iteration. Finally, since $|S|/(\log k)^{c''/\epsilon} \leq |S'_1| \leq |S|/2$, the last invariant also continues to hold. If $|S^*| > |S|/2$ also continues to hold, then all invariants hold, and we continue to the next iteration.

Assume now that $|S^*| \leq |S|/2$. In this case, we terminate the algorithm, and we return the current set E' of edges, set S''_0 of vertices of S, and we let $S''_1, \ldots, S''_q$ be all subsets of S lying in $\mathcal{S}$, together with the set S^*. Note that we are guaranteed that $|S^*| \geq |S|/4$, and, from the invariants, it is easy to verify that we obtain a valid output (except for the bound on $|E'|$ that we establish below). Note that the cardinality of the set S^* decreases by at least factor $(1 - 1/(\log k)^{c'/\epsilon})$ in each iteration, so the number of iterations in the algorithm is bounded by $O\left((\log k)^{c'/\epsilon+1}\right) \leq (\log k)^{4c'/\epsilon}$. Since the cardinality of edge set E' increases by at most $\frac{c'|S|D''\log^4 k}{\eta}$ in each iteration, we get that, at the end of the algorithm, $|E'| \leq \frac{|S|D''(\log k)^{c''/\epsilon}}{\eta}$. Lastly, since the running time for each iteration is $\widetilde{O}\left(k^{1+\epsilon}(\log k)^{O(1/\epsilon^2)}\right) + \widetilde{O}\left(|E(G)|D''\eta\right)$, the total running time of the algorithm is bounded by $\widetilde{O}\left(\left(k^{1+\epsilon}(\log k)^{O(1/\epsilon^2)} + |E(G)|D''\eta\right) \cdot (\log k)^{O(1/\epsilon)}\right)$. □

Lastly, the following corollary of Corollary E.4 allows us to either compute an expander X and embed it into $\mathcal{P}$ as before, or to compute a large collection of subsets of the input terminal set T that are far enough from each other.

Corollary E.5 *There is a constant c''', and a deterministic algorithm, that, given a graph G with non-negative lengths $\ell(e)$ on its edges $e \in E(G)$, a subset T of its vertices called terminals, with $|T| = k$ for some parameter k, together with parameters $\eta, D'', h \geq 1$ and $\epsilon \geq \Omega(1/\log k)$, such that $h \leq k/(100(\log k)^{c'''/\epsilon})$, computes one of the following:*

- *either a graph X with $V(X) \subseteq T$, and $|V(X)| \geq k/(64h)$, such that the maximum vertex degree in X is at most $O(\log k)$, and X is a φ-expander, for $\varphi = 1/(\log k)^{c'''/\epsilon}$, together with an embedding $\mathcal{P}$ of X into G, with congestion at most η, such that every path in $\mathcal{P}$ has length at most $64D''\log^2 k$; or*

- *a subset E' of at most $\frac{kD''(\log k)^{c'''/\epsilon}}{\eta}$ edges of G, and a collection $(T_1, \ldots, T_h)$ of disjoint subsets or T of cardinality at least $k/(h(\log k)^{c'''/\epsilon})$, such that for all $1 \leq i < i' \leq h$, in graph $G \setminus E'$, $\mathsf{dist}(T_i, T_{i'}) > D''$.*

The running time of the algorithm is $\widetilde{O}\left(\left(k^{1+\epsilon}(\log k)^{O(1/\epsilon^2)} + |E(G)|D''\eta\right) \cdot h(\log k)^{O(1/\epsilon)}\right)$.

Proof: The proof easily follows by iteratively applying the algorithm from Corollary E.4. We assume that $c''' \gg c''$, where c'' is the constant from Corollary E.4. The algorithm consists of $r \leq \lceil \log h \rceil + 2$ phases. For all $1 \leq j \leq r$, at the beginning of the jth phase, we are given a partition $\Pi^j = \left\{T^j_0, T^j_1, \ldots, T^j_{q_j}\right\}$ of the set T of terminals, and a set $E^j \subseteq E(G)$ of edges, such that the following hold:

- $|T^j_0| \leq jk/(4\log k)$;

- $|E^j| \leq \frac{kjD''(\log k)^{c''/\epsilon}}{\eta}$;

- for all $1 \leq i \leq q_j$, $\frac{k}{2^{j-1} \cdot (\log k)^{c''/\epsilon}} \leq |T^j_i| \leq k/2^{j-1}$; and

- for all $1 \leq i < i' \leq q_j$, $\mathsf{dist}_{G \setminus E^j}(T^j_i, T^j_{i'}) > D''$.

At the beginning of the algorithm, we set $T_0^1 = \emptyset$, $T_1^1 = T$, and $q_1 = 1$. We also set $E^1 = \emptyset$. It is easy to verify that all invariants hold.

We now describe the jth phase. At the beginning of the phase, we set $T_0^{j+1} = T_0^j$, $E^{j+1} = E^j$, and $\Pi^{j+1} = \emptyset$. We then perform q_j iterations, where in the ith iteration we process terminal set $T_i^j \in \Pi^j$.

We now describe an iteration for processing the terminal set T_i^j. If $|T_i^j| \le k/2^j$, then we simply add T_i^j to Π^{j+1}, set $T_0^{i,j+1} = \emptyset$, $E_i^{j+1} = \emptyset$, and continue to the next iteration. Assume now that $|T_i^j| > k/2^j$. Notice that in this case, from our assumption that $h \le k/(100(\log k)^{c'''/\epsilon})$, we get that $|T_i^j| \ge k/(4h) \ge 40(\log k)^{c''/\epsilon}$ holds. We then apply the algorithm from Corollary E.4 to graph $G \setminus E^j$, vertex set T_i^j, and the same parameters k, η, D'', ϵ as before. We now consider two cases. In the first case, the outcome of the algorithm from Corollary E.4 is a graph X with $V(X) \subseteq T_i^j$, and $|V(X)| \ge |T_i^j|/4 \ge k/2^{j+2} \ge k/(64h)$, such that the maximum vertex degree in X is at most $O(\log k)$, and X is a φ-expander, for $\varphi = 1/(\log k)^{c''/\epsilon} \ge 1/(\log k)^{c'''/\epsilon}$, together with an embedding $\mathcal{P}$ of X into G, with congestion at most η, such that every path in $\mathcal{P}$ has length at most $64D''\log^2 k$. We then terminate the algorithm, and return the expander X and its embedding $\mathcal{P}$.

Therefore, we assume from now on that the second case happens, and the algorithm from Corollary E.4 returns a subset E_i^{j+1} of at most $\frac{|T_i^j| D''(\log k)^{c''/\epsilon}}{\eta}$ edges of $G \setminus E^j$, and a partition $(S_0'', S_1'', S_2'', \dots, S_r'')$ of T_i^j into disjoint subsets, such that, for all $1 \le a \le z$, $|T_i^j|/(\log k)^{c''/\epsilon} \le |S_a''| \le |T_i^j|/2$, $|S_0''| \le |T_i^j|/(4\log k)$, and, in graph $G \setminus (E^j \cup E_i^{j+1})$, for all $1 \le i < i' \le z$, $\mathsf{dist}(S_i'', S_{i'}'') > D''$.

We set $T_0^{i,j+1} = S_0''$; note that $|T_0^{i,j+1}| \le |T_i^j|/(4\log k)$. Since $k/2^j \le |T_i^j| \le k/2^{j-1}$, we get that for all $1 \le a \le z$, $k/(2^j(\log k)^{c''/\epsilon}) \le |S_a''| \le k/2^j$. We then add all sets $S_1'', \dots, S_z''$ to Π^{j+1}, and continue to the next iteration.

The phase terminates when all sets in Π^j are processed. We set $E^{j+1} = E^j \cup (E_1^{j+1} \cup \dots \cup E_{q_j}^{j+1})$. Since, for all $1 \le i \le q_j$, $|E_i^{j+1}| \le \frac{|T_i^j| D''(\log k)^{c''/\epsilon}}{\eta}$, we get that $|E_1^{j+1} \cup \dots \cup E_{q_j}^{j+1}| \le \frac{kD''(\log k)^{c''/\epsilon}}{\eta}$, and so altogether, $|E^{j+1}| \le \frac{k(j+1)D''(\log k)^{c''/\epsilon}}{\eta}$. We also set $T_0^{j+1} = T_0^j \cup (T_0^{1,j+1} \cup \dots \cup T_0^{q_j,j+1})$. Since, for all $1 \le i \le q_j$, $|T_0^{i,j+1}| \le |T_i^j|/(4\log k)$, we get that $|T_0^{1,j+1} \cup \dots \cup T_0^{q_j,j+1}| \le k/(4\log k)$, and $|T_0^{j+1}| \le k(j+1)/(4\log k)$. From the above discussion, for all $T' \in \Pi^{j+1}$, $\frac{k}{2^j \cdot (\log k)^{c''/\epsilon}} \le |T'| \le k/2^j$ holds, and it is immediate to verify that for all $T', T'' \in \Pi^{j+1}$, $\mathsf{dist}_{G \setminus E^{j+1}}(T', T'') > D''$.

If the algorithm does not terminate with an expander X and its embedding $\mathcal{P}$, then it terminates after $r = \lceil \log h \rceil + 2$ phases. Since we are guaranteed that, for all $T' \in \Pi^r$, $|T'| \le k/2^r \le k/(2h)$, while $|T_0^r| \le rk/(4\log k) \le k/2$ we get that $|\Pi^r| \ge h$. We let $T_1, \dots, T_h$ be arbitrary h vertex sets in Π^r. We let $E' = E^r$; from our invariants, $|E'| \le \frac{kD''(\log k)^{c''/\epsilon+1}}{\eta} \le \frac{kD''(\log k)^{c'''/\epsilon}}{\eta}$. Clearly, for all $1 \le i < i' \le h$, in graph $G \setminus E'$, $\mathsf{dist}(T_i, T_{i'}) > D''$. Our invariants also guarantee that, for all $1 \le i \le h$, $|T_i| \ge \frac{k}{4h \cdot (\log k)^{c''/\epsilon}} \ge \frac{k}{(h(\log k)^{c'''/\epsilon})}$.

It now remains to analyze the running time of the algorithm. The algorithm consists of $O(\log k)$ phases, and each phase consists of $O(h)$ iterations. From Corollary E.4, the running time of each iteration is bounded by $\widetilde{O}\left(\left(k^{1+\epsilon}(\log k)^{O(1/\epsilon^2)} + |E(G)|D''\eta\right) \cdot (\log k)^{O(1/\epsilon)}\right)$. Therefore, the total running time of the algorithm is bounded by $\widetilde{O}\left(\left(k^{1+\epsilon}(\log k)^{O(1/\epsilon^2)} + |E(G)|D''\eta\right) \cdot h(\log k)^{O(1/\epsilon)}\right)$. □

We are now ready to complete the proof of Theorem 5.2. Recall that we are given as input a valid input structure $\mathcal{I} = \left(C, \{\ell(e)\}_{e \in E(C)}, D\right)$, where C is a connected graph, with arbitrary weights $w(x) \ge 0$ for vertices $x \in V(C)$, and parameters $0 < \epsilon < 1$, $\hat{D} > D$, and $\hat{W} \ge \max\left\{W(C), |E(C)|, 2^{\Omega(1/\epsilon)}\right\}$, together with a $(\hat{D}, \rho)$-pseudocut $\hat{E}$ for C of cardinality k, where $\rho = \hat{W}^{\epsilon}$.

We need to consider a special case, where $\epsilon < O(1/\log k)$, or, equivalently, $k < 2^{O(1/\epsilon)}$. In this case, since we have assumed that $\hat{W} > 2^{\Omega(1/\epsilon)}$, $k < \hat{W}^{\epsilon}$ holds. We then select an arbitrary edge $e \in \hat{E}$, and set $\hat{E}^* = \{e\}$. We let the expander X contain a single vertex t_e and no edges, and its embedding into $C_{|\hat{E}^*}$ is $\mathcal{P} = \emptyset$. We then return $\hat{E}^*, X$, and $\mathcal{P}$ and terminate the algorithm. Therefore, we assume from now on that $\epsilon \geq \Omega(1/\log k)$. We also assume that $k/(100(\log k)^{c'''/\epsilon}) \geq \left\lceil \hat{W}^{\epsilon} \right\rceil$, where c''' is the constant from Corollary E.5; otherwise, $k \leq \hat{W}^{2\epsilon}$ holds. In this case, we proceed as before: we select an arbitrary edge $e \in \hat{E}$, set $\hat{E}^* = \{e\}$, and return an expander X containing a single vertex t_e and no edges, and its embedding $\mathcal{P} = \emptyset$ into $C_{|\hat{E}^*}$. Therefore, we assume from now on that $k/(100(\log k)^{c'''/\epsilon} \geq \left\lceil \hat{W}^{\epsilon} \right\rceil$.

Let $G = C_{|\hat{E}}$ be the graph obtained from C by subdividing every edge $e = (u, v) \in \hat{E}$ with a vertex t_e. We denote the resulting edges by $e_1 = (u, t_e)$, $e_2 = (v, t_e)$, and we set the length of each of these edges to be $\ell(e)$. Let $T = \left\{ t_e \mid e \in \hat{E} \right\}$ be the set of the newly created vertices, that we refer to as *terminals*. Note that $k \leq \hat{W}$ must hold, since C is a connected graph and $\hat{W} \geq |E(C)|$.

We use the following parameters: $D'' = 40\hat{D}$, $h = \left\lceil \hat{W}^{\epsilon} \right\rceil$, and $\eta = 16D''\hat{W}^{\epsilon}(\log k)^{2c'''/\epsilon} \leq \hat{D} \cdot \hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}$. Note that, from our assumption, $h \leq k/(100(\log k)^{c'''/\epsilon})$ holds. We apply the algorithm from Corollary E.5 to graph G, the set T of terminals, parameters η, D'' and h that we just defined, and the input parameter ϵ. We now consider two cases, depending on the algorithm's outcome.

In the first, case, the algorithm from Corollary E.5 returns a graph X with $V(X) \subseteq T$, and $|V(X)| \geq k/(64h) \geq \Omega(k/\hat{W}^{\epsilon})$, such that the maximum vertex degree in X is at most $O(\log k) \leq O(\log \hat{W})$, and X is a φ^*-expander, for $\varphi^* = 1/(\log k)^{O(1/\epsilon)} \geq 1/(\log \hat{W})^{O(1/\epsilon)}$, together with an embedding $\mathcal{P}$ of X into G, with congestion at most $\eta \leq \hat{D} \cdot \hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}$, such that every path in $\mathcal{P}$ has length at most $64D'' \log^2 k \leq O(\hat{D} \log^2 \hat{W})$. We let $\hat{E}^* \subseteq \hat{E}$ be the set of edges corresponding to vertices of X, that is, $\hat{E}^* = \{e \mid t_e \in V(X)\}$. Then $|\hat{E}^*| = |V(X)| \geq \Omega(k/\hat{W}^{\epsilon})$. Moreover, the set $\mathcal{P}$ of paths that embeds X into G immediately defines a set $\mathcal{P}'$ of paths in graph $C_{|\hat{E}^*}$, embedding X into this graph with congestion at most $\eta \leq \hat{D} \cdot \hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}$, such that the length of every path in $\mathcal{P}$ is at most $O(\hat{D} \log^2 \hat{W})$ (note that graph G can be obtained from $C_{|\hat{E}^*}$ by subdividing each edge in $\hat{E} \setminus \hat{E}^*$, so the transformation from path set $\mathcal{P}$ to $\mathcal{P}'$ is straightforward). We then return the expander X, the edge set $\hat{E}^*$, and the embedding $\mathcal{P}'$ of X into $C_{|E^*}$ as the outcome of the algorithm.

From now on we focus on the second case, where the algorithm from Corollary E.5 returned a subset E' of edges, of cardinality at most:

$$\frac{kD''(\log k)^{c'''/\epsilon}}{\eta} \leq \frac{k}{16\hat{W}^{\epsilon}(\log k)^{c'''/\epsilon}}$$

(since $\eta = 16D''\hat{W}^{\epsilon}(\log k)^{2c'''/\epsilon}$), together with a collection $(T_1, \ldots, T_h)$ of disjoint subsets or T, such that the cardinality of each subset T_i is at least:

$$\frac{k}{h(\log k)^{c'''/\epsilon}} \geq \frac{k}{2\hat{W}^{\epsilon}(\log k)^{c'''/\epsilon}} \geq 8|E'|.$$

Recall that we are guaranteed that, for all $1 \leq i < i' \leq h$, $\mathsf{dist}_{G \setminus E'}(T_i, T_{i'}) > D'' = 40\hat{D}$. In the remainder of the proof, we will exploit the terminal sets $T_1, \ldots, T_h$, and the edge set E', in order to construct a $(\hat{D}, \rho)$-pseudocut $\hat{E}'$ in C, with $|\hat{E}'| \leq |\hat{E}| \left(1 - \frac{1}{\hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}}\right)$.

We define a set E'' of edges in graph C, as follows. For every edge $e \in E'$, if $e \in E(C)$, then we add e to E''. Otherwise, e was obtained by subdividing some edge $e' \in \hat{E}$. We then add e' to E''. Clearly, $|E''| \leq |E'|$. For all $1 \leq i \leq h$, we denote $\hat{E}_i = \left\{e \in \hat{E} \setminus E'' \mid t_e \in T_i\right\}$, and we let Z_i the set of all vertices of C that serve as endpoints of the edges in $\hat{E}_i$. Note that $|\hat{E}_i| \geq |T_i| - |E''| \geq |T_i|/2 \geq 4|E''|$. We need the following easy observation.

Observation E.6 *Let $C' = C \setminus E''$. Then for all $1 \leq i < j \leq h$, $\mathsf{dist}_{C'}(Z_i, Z_j) > 4\hat{D}$.*

Proof: Assume otherwise, and let $x \in Z_i, y \in Z_j$ be any pair of vertices with $\mathsf{dist}_{C'}(x, y) \leq 4\hat{D}$, and $i \neq j$. Let P be a shortest x-y path in graph C'. Let $e \in \hat{E}_i$ be the edge with endpoint x, and define $e' \in \hat{E}_j$ similarly. We claim that there is a path P' of length at most $10\hat{D}$ connecting t_e to $t_{e'}$ in graph $G \setminus E'$. Since $D'' = 40\hat{D}$, this will contradict the fact that $\mathsf{dist}_{G\setminus E'}(T_i, T_j) > D''$.

In order to obtain the path P, we start with the path P'; for every edge $\hat{e} \in E(P)$, if $\hat{e} \notin E(G)$, then we subdivide $\hat{e}$ to obtain a pair of edges that lie in G. Notice that this increases the length of the path by at most factor 2. We then append the edges (t_e, x) and $(y, t_{e'})$ to the beginning and the end of the resulting path, respectively. Since the length of every edge in C is at most $D \leq \hat{D}$, it is easy to verify that the length of P' is at most $10\hat{D}$. From our construction of edge set E'', and edge sets $\hat{E}_1, \ldots, \hat{E}_h$, it is also immediate to verify that no edge of P' lies in E'. □

For all $1 \leq i \leq h$, we let B_i the set of all vertices of C that can reach a vertex in Z_i via a path of length at most $2\hat{D}$ in graph $C \setminus E''$. Equivalently: $B_i = B_{C\setminus E''}(Z_i, 2\hat{D})$. Note that, from Observation E.6, for all $1 \leq i < j \leq h$, $B_i \cap B_j = \emptyset$. Therefore, there must be an index $1 \leq i \leq h$, such that the total weight of all vertices in B_i is bounded by $\frac{W(C)}{h} \leq \frac{W(C)}{\hat{W}^\epsilon}$. We fix this index i from now on.

We are now ready to define the final set $\hat{E}'$ of edges: $\hat{E}' = (\hat{E} \setminus \hat{E}_i) \cup E''$. Recall that $|\hat{E}_i| \geq |T_i|/2 \geq 4|E''|$, and $|\hat{E}_i| \geq \frac{k}{4\hat{W}^\epsilon (\log k)^{c'''/\epsilon}}$. Therefore, $|\hat{E}'| \leq |\hat{E}| - |\hat{E}_i| + |E''| \leq k\left(1 - \frac{1}{\hat{W}^\epsilon (\log k)^{O(1/\epsilon)}}\right) \leq k\left(1 - \frac{1}{\hat{W}^\epsilon (\log \hat{W})^{O(1/\epsilon)}}\right)$. It now remains to show that $\hat{E}'$ is a valid $(\hat{D}, \rho)$-pseudocut, for $\rho = \hat{W}^\epsilon$. We do so in the next claim.

Claim E.7 *Edge set $\hat{E}'$ is a $(\hat{D}, \rho)$-pseudocut for graph C.*

Proof: Consider any vertex $x \in V(C)$, and let $B = B_{C\setminus \hat{E}'}(x, \hat{D})$. It is enough to show that $W(B) \leq \frac{W(C)}{\hat{W}^\epsilon}$. We consider two cases.

Assume first that $B \cap Z_i \neq \emptyset$. In this case, $B \subseteq B_{C\setminus \hat{E}'}(Z_i, 2\hat{D}) \subseteq B_{C\setminus E''}(Z_i, 2\hat{D}) = B_i$ (since $E'' \subseteq \hat{E}'$). But then, from our choice of the index i, $W(B) \leq W(B_i) \leq \frac{W(C)}{\hat{W}^\epsilon}$.

Therefore, we assume that $B \cap Z_i = \emptyset$. Since $\hat{E} \setminus \hat{E}_i \subseteq \hat{E}'$, and no edge of $\hat{E}_i$ is incident to a vertex of B, if we consider the sub-graph H of $C \setminus \hat{E}'$ that is induced by B, then H is also a subgraph of $C \setminus \hat{E}$. In other words, $B \subseteq B_{C\setminus \hat{E}}(x, \hat{D})$. But then, since $\hat{E}$ was a valid $(\hat{D}, \rho)$-pseudocut for C, $W(B) \leq \frac{W(C)}{\hat{W}^\epsilon}$ must hold. □

It now remains to analyze the running time of the algorithm. In addition to running the algorithm from Corollary E.5, we need to compute the balls $B_1, \ldots, B_h$, and the final edge set $\hat{E}'$. Since the balls $B_1, \ldots, B_h$ are mutually disjoint, they can be computed in time $O(|E(C)|)$. Therefore, the running time is dominated by the running time of the algorithm from Corollary E.5, which is bounded by:

$$\widetilde{O}\left(h(\log k)^{O(1/\epsilon)} \cdot \left(k^{1+\epsilon}(\log k)^{O(1/\epsilon^2)} + |E(G)|D''\eta\right)\right) \leq O\left(|E(C)| \cdot \hat{D}^2 \cdot \hat{W}^{O(\epsilon)}(\log \hat{W})^{O(1/\epsilon^2)}\right).$$

since $h \leq 2W^{\epsilon}$, $D'' = \Theta(D)$, $\eta = \hat{D} \cdot \hat{W}^{\epsilon}(\log \hat{W})^{O(1/\epsilon)}$, $k \leq |E(C)| \leq \hat{W}$, and $|E(G)| \leq O(|E(C)|)$.

In order to complete the proof of Theorem 5.2, it is now enough to prove Lemma E.1, which we do next.

E.1.1 Proof of Lemma E.1

The proof follows standard techniques, namely the Cut-Matching game of Khandekar, Rao, and Vazirani [KRV09], and more specifically, its variant that was studied in [KKOV07], and then used in [CGL+19]. At a high level, the Cut-Matching game is a game played between two players: a cut player and a matching player. The game starts with a graph X containing an even number n of vertices and no edges, and then proceeds in iterations. In every iteration i, a cut player computes some partition (A_i, B_i) of $V(X)$, with $|A_i| \leq |B_i|$. The matching player then returns a matching M_i between vertices of A_i and vertices of B_i, of cardinality $|A_i|$. The edges of M_i are added to the graph X, and the iteration terminates. Once graph X becomes a φ-expander, for some given parameter φ, the game terminates. Intuitively, the goal of the cut player is to make the game as short as possible, while the matching player wants to make it longer. We will use the following theorem, that was proved in [CGL+19] in order to implement the cut player.

Theorem E.8 (Theorem 4.1 in [CGL+19]) *There are universal constants c_0, N_0 and a deterministic algorithm, that, given an n-vertex graph $G = (V, E)$ and parameters N, q with $N > N_0$ an integral power of 2, and $q \geq 1$ an integer, such that $n \leq N^q$, and the maximum vertex degree in G is at most $c' \log n$ for a constant c', computes one of the following:*

- *either a partition (A, B) of V with $|A|, |B| \geq n/4$, $|A| \leq |B|$, and $|E_G(A, B)| \leq n/100$; or*
- *a subset $R \subseteq V$ of at least $n/2$ vertices of G, such that graph $G[R]$ is a φ'-expander, for $\varphi' = 1/(q \log N)^{8q}$.*

The running time of the algorithm is $O\left(N^{q+1} \cdot (q \log N)^{c_0 q^2}\right)$.

We will use the following immediate corollary of the theorem.

Corollary E.9 *There is a universal constant c'', and a deterministic algorithm, that, given an n-vertex graph $G = (V, E)$, and a parameter $0 < \epsilon < 1$, such that $\epsilon \geq c''/\log n$, and the maximum vertex degree in G is at most $c' \log n$ for some constant c', computes one of the following:*

- *either a partition (A, B) of $V(G)$ with $|A|, |B| \geq n/4$, $|A| \leq |B|$, and $|E_G(A, B)| \leq n/100$; or*
- *a subset $R \subseteq V$ of at least $n/2$ vertices of G, such that graph $G[R]$ is a φ''-expander, for $\varphi'' = 1/(\log n)^{O(1/\epsilon)}$.*

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

Proof: We set $q = \lceil 1/\epsilon \rceil$, so that $1/\epsilon \leq q \leq 2/\epsilon$. Next, we let N be the smallest integral power of 2, so that $N^q \geq n$. It is easy to verify that $N \leq 2n^{1/q} \leq 2n^{\epsilon}$. Also, $N \geq n^{1/q} \geq n^{\epsilon/2}$. We can then set the constant c'' to be large enough to ensure that $n^{\epsilon/2} > N_0$, where N_0 is the constant in the statement of Theorem E.8. We then apply Theorem E.8 to the graph G, together with parameters q and N. If the algorithm returns a partition (A, B) of $V(G)$ with $|A|, |B| \geq n/4$ and $|E_G(A, B)| \leq n/100$, then we also return this partition as the outcome of our algorithm. Otherwise, the algorithm from Theorem E.8 computes a subset $R \subseteq V$ of at least $n/2$ vertices of G, such that graph $G[R]$ is a φ''-expander, for $\varphi'' = 1/(q \log N)^{8q}$. Since $q = O(1/\epsilon)$, and $N^q \leq 2n$, it is easy to verify that $\varphi'' \geq 1/(\log n)^{O(q)} \geq 1/(\log n)^{O(1/\epsilon)}$. We then return R as the outcome of the algorithm.

Lastly, the running time of the algorithm is $O\left(N^{q+1} \cdot (q \log N)^{c_0 q^2}\right)$. Since $N^q \leq 2n$ and $q = \Theta(1/\epsilon)$, we get that the running time is bounded by $O\left(n^{1+O(\epsilon)} \cdot (\log n)^{O(1/\epsilon^2)}\right)$. □

Consider now the following implementation of the cut-matching game. We start with a graph X that contains an even number n of vertices and no edges, and play the game for at most $c' \log n$ iterations, where c' is the constant from Corollary E.9. In every iteration i, we apply the algorithm from Corollary E.9 to the graph X, with the parameter ϵ. We now consider two cases. First, if the outcome of the algorithm from Corollary E.9 is a partition (A, B) of $V(G)$ with $|A|, |B| \geq n/4$, $|A| \leq |B|$, and $|E_G(A, B)| \leq n/100$, then we let (A_i, B_i) be any partition of $V(G)$ with $|A_i| = |B_i|$, such that $A \subseteq A_i$. We then compute an arbitrary perfect matching M_i between vertices of A_i and vertices of B_i. Lastly, we add the edges of M_i to graph X, and continue to the next iteration. In the second case, the outcome of the algorithm from Corollary E.9 is a subset $R \subseteq V$ of at least $n/2$ vertices of G, such that graph $G[R]$ is a φ''-expander, for $\varphi'' = 1/(\log n)^{O(1/\epsilon)}$. In this case, we let $A_i = V \setminus R$, $B_i = R$, and we compute an arbitrary matching M_i of cardinality $|A_i|$ between vertices of A_i and vertices of B_i. We then add the edges of M_i to graph X and terminate the algorithm.

The following theorem follows immediately from the results of [KKOV07], and was also proved explicitly in [CGL+19] (see Theorem 2.5 in [CGL+19]).

Theorem E.10 *There is a constant $\hat{c}$, such that, after at most $\hat{c} \log n$ iterations, the above algorithm is guaranteed to terminate, and the resulting graph X is a $\hat{\varphi}$-expander, for $\hat{\varphi} = 1/(\log n)^{\hat{c}/\epsilon}$.*

Notice that the above theorem holds regardless of the specific choices of the matchings M_i in each iteration (though the partitions (A_i, B_i) of the vertices of V that the algorithm computes in each iteration depend on the choices of the matchings from previous iterations). We note that the constant c' in the statement of Theorem E.8 from [CGL+19] was chosen to be such that $\hat{c} \leq c'$ holds.

The following theorem will be used in order to implement the matching player. The proof uses techniques that are similar to those introduced in [CK19], and then used in [CGL+19] and [CS21].

Theorem E.11 *There is a deterministic algorithm, that, given a graph G with integral lengths $\ell(e) \geq 1$ on its edges $e \in E(G)$, two disjoint subsets A, B of its vertices with $|A| \leq |B|$ and $|A| = \kappa$, together with parameters $D', \eta > 0$, and $z \geq 0$, computes one of the following:*

- *either a collection $\mathcal{P}$ of at least $|A| - z$ paths in G, where each path connects a distinct vertex of A to a distinct vertex of B; every path has length at most D'; and every edge of G participates in at most η paths; or*
- *a collection E' of at most $\frac{|A| \cdot D' \log \kappa}{2\eta}$ edges of G, and two subsets $A^* \subseteq A$, $B^* \subseteq B$ of at least $z/2$ vertices each, such that, in graph $G \setminus E'$, $\mathsf{dist}(A^*, B^*) > D'$.*

The running time of the algorithm is $\widetilde{O}(|E(G)| D' \eta)$.

We delay the proof of Theorem E.11 to Appendix E.1.2, after we complete the proof of Lemma E.1 using it.

In the remainder of the proof, we let $\hat{c}$ be the constant from Theorem E.10, so that the number of iterations in the Cut-Matching game played over a set of k vertices is bounded by $\hat{c}\log k$, and the final graph X obtained at the end of the game is a $\hat{\varphi}$-expander, for $\hat{\varphi} = 1/(\log k)^{\hat{c}/\epsilon}$. The algorithm consists of two stages. In the first stage, we compute an expander X over the vertex set S, and a relatively small subset $F \subseteq E(X)$ of its edges (that we call *fake edges*), together with an embedding of $X \setminus F$ into G; if we fail to compute this expander, then we will produce the desired set E' of at most $\frac{ckD'\log^2 k}{\eta}$ edges, and two disjoint subsets $S_1, S_2 \subseteq S$ of vertices of cardinality at least $k/(\log k)^{c/\epsilon}$ each, such that, in graph $G \setminus E'$, $\mathsf{dist}(S_1, S_2) > D'$. In the second stage, we use expander pruning in order to get rid of the fake edges and compute a final expander $X' \subseteq X \setminus F$. We now describe each stage in turn.

Stage 1: Embedding the Expander. We start with the graph X, whose vertex set is S, and edge set is empty. We also let $\mathcal{P} = \emptyset$ be an initial embedding of the graph X. We then perform at most $\hat{c}\log k$ iterations, with the ith iteration performed as follows. First, we apply the algorithm from Corollary E.9 to graph X. Assume first that the algorithm produces a partition (A_i, B_i) of $V(X)$ with $|A_i|, |B_i| \geq k/4$, $|A_i| \leq |B_i|$, and $|E_X(A_i, B_i)| \leq k/100$. We let (A'_i, B'_i) be any partition of $V(X)$ with $|A'_i| = |B'_i|$ and $A_i \subseteq A'_i$.

Next, we apply the algorithm from Theorem E.11 to graph C, the sets A'_i, B'_i of its vertices, with the same parameter D', congestion parameter $\hat{\eta} = \eta/(\hat{c}\log k)$, and $z = \hat{\varphi}k/(20\hat{c}^2\log^2 k)$. Assume first that the algorithm returns a collection E' of at most $\frac{|A'_i|\cdot D'\log k}{2\hat{\eta}} \leq \frac{\hat{c}|A'_i|\cdot D'\log^2 k}{2\eta} = \frac{\hat{c}kD'\log^2 k}{4\eta}$ edges of G, and two subsets $A^* \subseteq A_i$, $B^* \subseteq B_i$ of at least $z/2 = \hat{\varphi}k/(40\hat{c}^2\log^2 k)$ vertices each, such that, in graph $G \setminus E'$, $\mathsf{dist}(A^*, B^*) > D'$. We then say that the current iteration was unsuccessful, terminate the algorithm, and return the sets $S_1 = A^*, S_2 = B^*$ of vertices, together with the set E' of edges. It is easy to verify that they have all required properties, since $\hat{\varphi}k/(40\hat{c}^2\log^2 k) = k/(\log k)^{O(1/\epsilon)}$.

Assume now that the algorithm returned a collection $\mathcal{P}_i$ of at least $|A'_i| - z$ paths in G, where each path connects a distinct vertex of A'_i to a distinct vertex of B'_i, every path has length at most D', and every edge of G participates in at most $\hat{\eta} = \eta/(\hat{c}\log k)$ paths. We then say that the current iteration is successful. We let M'_i be a partial matching between vertices of A'_i and vertices of B'_i, where $(x, y) \in M'_i$ iff some path in $\mathcal{P}_i$ connects x to y. Let $A''_i \subseteq A'_i, B''_i \subseteq B'_i$ be the sets of vertices that do not participate in the matching M'_i. We let F_i be an arbitrary perfect matching between vertices of A''_i and vertices of B''_i; observe that $|F_i| = |A'_i \setminus A''_i| \leq z$ must hold. We call the vertex pairs in F_i *fake edges for graph* X. We then let $M_i = M'_i \cup F_i$, and add the edges of M_i into X, terminating the current iteration.

Finally, assume that the algorithm from Corollary E.9 returns a subset $R \subseteq V(X)$ of at least $k/2$ vertices of X, such that graph $X[R]$ is a φ-expander. Let i be the index of the current iteration. We set $A'_i = V(X) \setminus R$ and $B'_i = R$, and apply the algorithm from Theorem E.11 to graph G, vertex sets A'_i, B'_i, distance bound D', congestion bound $\hat{\eta} = \eta/(\hat{c}\log k)$, and parameter $z = \hat{\varphi}k/(20\hat{c}^2\log^2 k)$ as before. We then continue exactly as before. If the algorithm returns a collection E' of at most $\frac{\hat{c}|A'_i|\cdot D'\log^2 k}{2\eta} \leq \frac{\hat{c}kD'\log^2 k}{4\eta}$ edges of G, and two subsets $A^* \subseteq A_i$, $B^* \subseteq B_i$ of at least $z/2 = \hat{\varphi}k/(40\hat{c}^2\log^2 k)$ vertices each, such that, in graph $G \setminus E'$, $\mathsf{dist}(A^*, B^*) > D'$, then, as before, we say that the current iteration was unsuccessful, terminate the algorithm, and return the sets $S_1 = A^*, S_2 = B^*$ of vertices, together with the set E' of edges. Otherwise, the algorithm from Theorem E.11 returns a collection $\mathcal{P}_i$ of at least $|A'_i| - z$ paths in G, where each path connects a distinct vertex of A'_i to a distinct vertex of B'_i, every path has length at most D', and every edge of G participates in at most $\hat{\eta} = \eta/(\hat{c}\log k)$ paths. We then say that the current iteration is successful. We let M'_i be a partial matching between vertices of A'_i and vertices of B'_i, where $(x, y) \in M'_i$ iff some path in $\mathcal{P}_i$ connects x to y. Let $A''_i \subseteq A'_i, B''_i \subseteq B'_i$

be the sets of vertices that do not participate in the matching M'_i. We let F_i be an arbitrary matching between vertices of A''_i and vertices of B''_i of cardinality $|A''_i|$; observe that $|F_i| = |A'_i \setminus A''_i| \leq z$ must hold. We call the vertex pairs in F_i *fake edges for graph X*. We then let $M_i = M'_i \cup F_i$, and add the edges of M_i into X, terminating the current the algorithm.

If any iteration of the algorithm was unsuccessful, then we have obtained the desired sets S_1, S_2 of vertices of S, and the edge set E' as required. We assume now that every iteration of the algorithm was successful. Then, from Theorem E.10, the total number of iterations in the algorithm is bounded by $\hat{c} \log k$, and the final graph X is a $\hat{\varphi}$-expander, for $\hat{\varphi} = 1/(\log k)^{\hat{c}/\epsilon}$. Let $F = \bigcup_i F_i$ be the set of all fake edges of X. Then $|F| \leq \hat{c} z \log k$. Let $\mathcal{P} = \bigcup_i \mathcal{P}_i$. Then the paths in $\mathcal{P}$ provide an embedding of the graph $X \setminus F$ into G, where the length of every path in $\mathcal{P}$ is at most D'. Since the number of iterations is at most $\hat{c} \log k$, and each set $\mathcal{P}_i$ of paths causes edge-congestion at most $\eta/(\hat{c} \log k)$, the total edge-congestion caused by the paths in $\mathcal{P}$ is at most η. This finishes the first stage of the algorithm. We now analyze its running time.

The algorithm consists of $O(\log k)$ iterations. Each iteration involves applying the algorithm from Corollary E.9, whose running time is $O\left(k^{1+O(\epsilon)} \cdot (\log k)^{O(1/\epsilon^2)}\right)$, and the algorithm from Theorem E.11, whose running time is $\widetilde{O}(|E(G)|D'\hat{\eta}) \leq \widetilde{O}(|E(G)|D'\eta)$. Therefore, the total running time of the algorithm is:

$$\widetilde{O}\left(k^{1+O(\epsilon)} \cdot (\log k)^{O(1/\epsilon^2)}\right) + \widetilde{O}(|E(G)|D'\eta).$$

Stage 2: Pruning. In this stage, we apply the algorithm from Theorem 2.2 to the graph X, with expansion parameter $\hat{\varphi} = 1/(\log k)^{\hat{c}/\epsilon}$, and the sequence σ containing the edges of F in an arbitrary order. Observe that the maximum vertex degree in X is at most $\hat{c} \log k$, and $|F| \leq (\hat{c} \log k) \cdot z \leq (\hat{c} \log k) \cdot \hat{\varphi} k/(20\hat{c}^2 \log^2 k) \leq \hat{\varphi} k/(20\hat{c} \log k)$. Clearly, if Δ denotes the maximum vertex degree in X, then $|F| \leq \hat{\varphi}|E(X)|/(10\Delta)$ holds. Let $S' = S_{|F|}$ be the outcome of the algorithm, and let X' be the graph obtained from $X \setminus F$ after the vertices of S' are removed from it. Then, from Theorem 2.2 graph X' is a φ-expander, for $\varphi = \hat{\varphi}/(6\Delta) = 1/(\log k)^{O(1/\epsilon)}$. Let $\mathcal{P}' \subseteq \mathcal{P}$ be the set of paths corresponding to the edges in X'. Then paths in $\mathcal{P}'$ define an embedding of X' into G with congestion at most η, where each path has length at most D'. Recall also that Theorem 2.2 guarantees that $|S'| \leq 8|F|\Delta/\hat{\varphi} \leq 8k/20$. Therefore, $|V(X')| \geq |S|/2$ must hold.

The running time of the second stage of the algorithm is bounded by $\widetilde{O}(|F|\Delta^2/\hat{\varphi}^2) \leq \widetilde{O}(k \log^2 k/\hat{\varphi}) \leq \widetilde{O}\left(k \cdot (\log k)^{O(1/\epsilon)}\right)$. The total running time of the whole algorithm is therefore bounded by:

$$\widetilde{O}\left(k^{1+O(\epsilon)} (\log k)^{O(1/\epsilon^2)}\right) + \widetilde{O}(|E(G)|D'\eta).$$

In order to complete the proof of Theorem 5.2, it is now enough to prove Theorem E.11, which we do next.

E.1.2 Proof of Theorem E.11

The proof proceeds by iteratively applying the following claim.

Claim E.12 *There is a deterministic algorithm, that, given a graph G with integral lengths $\ell(e) \geq 1$ on its edges $e \in E(G)$, two disjoint subsets A', B' of its vertices with $|A'| \leq |B'|$, together with parameters $D' > 0$ and $\eta' > 0$, computes one of the following:*

- *either a collection $\mathcal{P}'$ of at least $|A'|/2$ paths in G, where each path connects a distinct vertex of A' to a distinct vertex of B', and every path has length at most D', and every edge in G participates in at most η' paths; or*

- *a collection E' of at most $|A'| \cdot D'/(2\eta')$ edges of G, and two subsets $A'' \subseteq A'$, $B'' \subseteq B'$ of at least $|A'|/2$ vertices each, such that, in graph $G \setminus E'$, $\mathsf{dist}(A'', B'') > D'$.*

The running time of the algorithm is $\widetilde{O}(|E(G)|D'\eta')$.

We provide the proof of Claim E.12 below, after we complete the proof of Theorem E.11 using it. We start with $\mathcal{P} = \emptyset$, set $A' = A, B' = B$, and then iterate, as long as $|A'| > z$. In every iteration, we apply the algorithm from Claim E.12 to the input graph G, parameter D', and $\eta' = \eta/\log\kappa$. If the outcome of the algorithm is a set E' of edges, of cardinality at most $\frac{|A'|\cdot D'}{2\eta'} \leq \frac{|A|\cdot D'\log k}{2\eta}$, and two subsets $A'' \subseteq A'$, $B'' \subseteq B'$ of at least $|A'|/2$ vertices each, such that, in graph $G \setminus E'$, $\mathsf{dist}(A'', B'') > D'$, then we set $A^* = A'', B^* = B''$, and terminate the algorithm with the output A^*, B^* and E'. Note that, since $|A'| \geq z$ held, we are guaranteed that $|A^*|, |B^*| \geq z/2$, as required. Otherwise, we obtain a collection $\mathcal{P}'$ of at least $|A'|/2$ paths in G, where each path connects a distinct vertex of A' to a distinct vertex of B', and every path has length at most D', and every edge in G participates in at most η' paths in $\mathcal{P}'$. We add the paths in $\mathcal{P}'$ to the set $\mathcal{P}$, and we delete from A' and B' vertices that serve as endpoints to paths in $\mathcal{P}'$, terminating the current iterations.

If any iteration terminates with the set E' of edges, then the algorithm produces the required output. Therefore, we assume from now on that in every iteration, new paths are added to set $\mathcal{P}$. Since the cardinality of the vertex set A' must decrease by at least factor 2 in every iteration, the number of iterations is bounded by $\log\kappa$. Since each set $\mathcal{P}'$ of paths computed by the algorithm from Claim E.12 causes edge-congestion at most $\eta' = \eta/\log\kappa$, the paths in the final set $\mathcal{P}$ cause edge-congestion at most η. Clearly, the length of every path in $\mathcal{P}$ is at most D', and the endpoints of all paths in $\mathcal{P}$ are distinct. Moreover, when the algorithm terminates, $|A'| \leq z$ holds, so we are guaranteed that $|\mathcal{P}| \geq \kappa - z$. Since the running time of every iteration is $\widetilde{O}(|E(G)|D'\eta') = \widetilde{O}(|E(G)|D'\eta/\log\kappa)$, and the total number of iterations is bounded by $\log\kappa$, the total running time is at most $\widetilde{O}(|E(G)|D'\eta)$.

In order to complete the proof of Theorem E.11, it is now enough to prove Claim E.12, which we do next.

Proof of Claim E.12. For convenience, we denote $|A'| = \kappa$. We construct a new graph H: start with the graph G, and add a source vertex s that connects to every vertex in A' with an edge of length 1; similarly, add a destination vertex t, that connects to every vertex in B' with an edge of length 1. All other edge lengths remain unchanged in H. We use the standard Even-Shiloach Tree data structure on graph H, with source vertex s, and distance threshold $D' + 2$. Initialize $\mathcal{P}' = \emptyset$. Additionally, for every edge $e \in E(G)$, we maintain a value $\gamma(e)$ – the number of paths in $\mathcal{P}'$ that contain the edge e. Initially, $\gamma(e) = 0$ for all $e \in E(G)$.

While the distance from s to t is less than $D' + 2$, choose any path P in H connecting s to t of length at most $D' + 2$. Let P' be the path obtained from P after we delete its endpoints, so $\ell(P') \leq D'$. Add path P' to $\mathcal{P}'$, and update, for every edge $e \in E(P')$, the value $\gamma(e)$. If $\gamma(e)$ exceeds η', then we remove edge e from graph H. Additionally, we remove from H the fist and the last edge of P (the edges that are incident to s and t). We then continue to the next iteration. The total update time of the data structure is $\widetilde{O}(|E(G)|D')$, and the total running time of the algorithm, that includes selecting the paths and deleting edges from G as required, is bounded by $\widetilde{O}(|E(G)|D'\eta')$, since an edge may participate in at most η' paths in $\mathcal{P}'$. We now consider two cases. First, if $|\mathcal{P}'| \geq |A'|/2$ at the end of the algorithm, then we terminate the algorithm, and return the set $\mathcal{P}'$ of paths. Clearly, the paths in $\mathcal{P}'$ cause edge-congestion at most η', and the length of every path is at most D'.

From now on, we assume that, when the algorithm terminates, $|\mathcal{P}'| < |A'|/2$ holds. Let E' denote the set of all edges of graph G that were deleted from H. Let $A'' \subseteq A', B'' \subseteq B'$ be the sets of vertices that do not serve as endpoints of paths in $\mathcal{P}'$. Then $\mathsf{dist}_{G \setminus E'}(A'', B'') > D'$ (as otherwise we could have continued the algorithm). Moreover, an edge belongs to E' only if η' paths in $\mathcal{P}'$ contain this edge. Since each path in $\mathcal{P}'$ contains at most D' edges, (as the length of each pedge is at least 1), $|E'| \leq |\mathcal{P}'| \cdot D'/\eta' \leq |A'| \cdot D'/(2\eta')$. □

E.2 Proof of Theorem 5.3

The proof is almost identical to that of [CS21]. Our main tool is the following theorem that allows us to embed a small expander into a large expander. The theorem is almost identical to Theorem 3.8 in [CS21], except that we need a slightly different tradeoff between various parameters and the running time. We include the proof in Appendix E.2.1 for completeness.

Lemma E.13 (Analogue of Theorem 3.8 in [CS21]) *There is a constant c and a deterministic algorithm that, given an n-vertex graph X that is a φ-expander with maximum vertex degree Δ, and a set $T \subseteq V(X)$ of its vertices whose cardinality is k, together with a parameter $0 < \epsilon < 1$, such that $\varphi \leq 1/2^{\Omega(1/\epsilon)}$, computes a graph X' with $V(X') = T$ that has maximum vertex degree at most $c \log k$, such that graph X' is a $\hat{\varphi}$-expander, for $\hat{\varphi} = 1/(\log k)^{c/\epsilon}$. The algorithm also computes an embedding $\mathcal{P}$ of X' into X, such that the paths in $\mathcal{P}$ have length at most $\frac{c\Delta \log n}{\varphi}$ and cause congestion at most $\frac{c\Delta^2 \log^3 n}{\varphi^2}$. The running time of the algorithm is $\widetilde{O}(\Delta^4 n/\varphi^3) + O\left(k^{1+O(\epsilon)} \cdot (\log k)^{O(1/\epsilon^2)}\right)$.*

The proof of Lemma E.13 is deferred to Appendix E.2.1. We now turn to complete the proof of Theorem 5.3 using it.

Throughout the algorithm, we denote $n = V(X)$. We use the parameters $\gamma = \lceil n^\epsilon \rceil$ and $r = \lfloor 1/\epsilon \rfloor$. The idea is to construct and maintain a hierarchy $X_0, X_1, \ldots, X_r$ of expanders, where $X_0 = X$, and for $i > 0$, $|V(X_i)| = \lceil n/\gamma^i \rceil$. For all i, we will ensure that $V(X_i) \subseteq V(X_{i-1})$, and we will maintain an embedding of X_i into X_{i-1} via short paths that cause small edge-congestion.

Specifically, we will use the following parameters. We let c be the constant from Lemma E.13, that we can assume to be a large enough constant. We set $\varphi_0 = \varphi$, and, for $1 \leq i \leq r$, $\varphi_i = \hat{\varphi} = 1/(\log n)^{c/\epsilon}$. We also denote $\Delta_0 = \Delta$, and for all $1 \leq i \leq r$, $\Delta_i = c \log n$. We will ensure that for all $0 \leq i \leq r$, X_i is a φ_i-expander, whose maximum vertex degree is at most Δ_i. We also define parameters $\ell_1 = c\Delta \log n/\varphi$, and for $1 < i \leq r+1$, $\ell_i = \hat{\ell} = c\Delta_{i-1} \log n/\varphi_{i-1} = c^2 \log^2 n/\hat{\varphi}$. Additionally, we define parameters $\eta_1 = c\Delta^2 \log^3 n/\varphi^2$, and for $1 < i \leq r+1$, $\eta_i = c\Delta_{i-1}^2 \log^3 n/\varphi_{i-1}^2 = c^3 \log^5 n/\hat{\varphi}^2$.

For all $1 \leq i < r$, we will maintain an embedding $\mathcal{P}_i$ of X_i into X_{i-1}, where the paths in $\mathcal{P}_i$ have length at most ℓ_i each, and cause total congestion at most η_i. Additionally, we will maintain an ES-Tree data structure τ_i in graph X'_i, that is obtained from X_i by adding a source vertex s_i, and connecting it to all vertices in $V(X_{i+1})$. The tree is rooted at s_i, and its depth bound is $D_i = 4\ell_i$. Lastly, for every edge $e \in E(X_{i-1})$, we will maintain a list $J_i(e)$ of all edges $e' \in E(G_i)$, such that the embedding of e' in $\mathcal{P}_i$ contains the edge e; recall that $|J(e)| \leq \eta_i$ must hold. Whenever edge e is deleted from graph X_{i-1}, this will trigger the deletion of all edges in its list $J_i(e)$ from graph X_i. We use the algorithm from Theorem 2.2 in order to maintain, for every expander X_i, the set S_i of "pruned-out" vertices. When set S_i becomes too large, we re-initialize the graphs $X_i, X_{i+1}, \ldots, X_r$, and all the corresponding data structure.

The outcome of the algorithm (the vertex set S that we maintain) is the set $S = S_0$ of vertices – the vertices that we prune out of the main expander $X_0 = X$.

Algorithm InitializeExpander(i)

Assumption: graph X_{i-1} is defined; $|V(X_{i-1})| \geq n/(2\gamma^{i-1})$; graph X_{i-1} is a $\varphi_{i-1}/(6\Delta_{i-1})$-expander, with maximum vertex degree is at most Δ_{i-1}.

- If $i = r+1$, then initialize an ES-Tree τ_r in graph X_r, rooted at an arbitrary vertex, with distance threshold D_{r+1}; return.
- Let V_i be an arbitrary subset of $V(X_{i-1})$, of cardinality $\lceil n/\gamma^i \rceil$.
- Apply the algorithm from Lemma E.13, to graph X_{i-1}, with $T = V_i$, to compute an expander X_i with vertex set V_i, and its embedding $\mathcal{P}_i$ into X_{i-1}, so that X_i is a φ_i-expander, with maximum vertex degree at most Δ_i; the paths in $\mathcal{P}_i$ cause edge-congestion at most η_i and have length at most ℓ_i each.
- For every edge $e \in E(X_{i-1})$, initialize the list $J_i(e)$ of all edges of X_i whose embedding path in $\mathcal{P}_i$ contains e.
- Initialize the expander pruning algorithm from Theorem 2.2 on graph X_i, that will maintain a pruned vertex set $S_i \subseteq V(X_i)$.
- Initialize an ES-tree τ_{i-1} in graph X'_{i-1} that is obtained from X_{i-1} by adding a source vertex s_{i-1} and connecting it to all vertices in $V(X_i)$. The tree τ_{i-1} is rooted at s_{i-1} and has depth threshold D_i.
- Call InitializeExpander($i+1$).

Figure 1: Algorithm InitializeExpander(i)

Initializing the Data Structures

At the beginning of the algorithm, we run procedure InitializeExpander(1), that constructs expander X_1, its embedding $\mathcal{P}_1$ into X_0, and the lists $J_1(e)$ for edges $e \in E(X_0)$. The procedure then recursively calls to InitializeExpander(2), that constructs the data structures for higher levels. The algorithm for procedure InitializeExpander(i) is described in Figure 1. We note that it is identical to the algorithm of [CS21]. We emphasize that procedure InitializeExpander(i) is only called when graph X_{i-1} is defined, with $|V(X_{i-1})| \geq n/(2\gamma^{i-1})$, and we are guaranteed that X_{i-1} is a $\varphi_{i-1}/(6\Delta_{i-1})$-expander, whose maximum vertex degree is at most Δ_{i-1}. From Observation 2.1, every pair of vertices of X_{i-1} is then guaranteed to have a path of length at most $\frac{8\Delta_{i-1}\log n}{\varphi_{i-1}} \leq \ell_i \leq D_i/2$, so throughout the algorithm, tree τ_{i-1} spans all vertices of X_{i-1}.

We note that we will ensure that, over the course of the algorithm, for all $1 \leq i \leq r$, the size of set S_i never exceeds $n/(2\gamma^i)$.

Maintaining the Data Structures

For all $0 \leq i < r$, we denote by $X^{(0)}_{i+1}$ the expander graph created by Procedure InitializeExpander(i). For all $t > 0$, we denote by $X^{(t)}_{i+1}$ the graph that is obtained from $X^{(0)}_{i+1}$ after t edge deletions from X. As t increases, our algorithm maintains the graph $X_{i+1} = X^{(t)}_{i+1} \setminus S_{i+1}$. By Theorem 2.2, as long as $t \leq \varphi_{i+1}|E(X_{i+1})|/(20\Delta_{i+1})$, graph X_{i+1} remains a $(\varphi_{i+1}/(6\Delta_{i+1}))$-expander, and $|V(X_{i+1})| \geq |V(X^{(0)}_{i+1})|/2 \geq n/(2\gamma^i)$.

```
ALGORITHM DeleteEdge($(i, e)$)

Assumption: edge $e$ lies in graph $X_i$.

• If $i = r$, delete $e$ from graph $X_r$. Recompute the ES-Tree $\tau_r$ in graph $X_r$, rooted
  at any vertex, with depth threshold $D_{r+1}$; return.

• Delete edge $e$ from graph $X_i$ and from the data structure $\tau_i$.

• Update the pruned-out vertex set $S_i$ using the algorithm from Theorem 2.2, and
  update the tree $\tau_{i-1}$, by deleting edges $(s, x)$ for every vertex $x$ that was added to
  $S_i$.

• If the total number of edge deletions from $X_i^{(0)}$ exceeds $\varphi_i|E(X_i^{(0)})|/(20\Delta_i)$, call
  InitializeExpander($(i-1)$); return. (Note: here we only count edges that were deleted
  from $X$ as part of input edge-deletion sequence, and we do not include edges that
  are incident to vertices of $S_i$).

• Let $Z_i^{new}$ denote the set of edges that were just removed from $X_i$. That is, $Z_i^{new}$
  contains the edge $e$ and all edges incident to vertices that were just added to $S_i$.

• For each edge $e \in Z_i^{new}$, for every edge $e' \in J_i(e)$, call Delete($(i+1, e')$).
```

Figure 2: Algorithm DeleteEdge(i, e)

When some edge e is deleted from graph X, we call Algorithm DeleteEdge$(0, e)$. The algorithm may recursively call to procedure DeleteEdge(i, e') for other expander graphs X_i and edges e'. The algorithm DeleteEdge(i, e) is shown in Figure 2. We assume that edge e lies in graph X_i.

Total Update Time

For all $0 \le i \le r$, we denote $n_i = \lceil n/\gamma^i \rceil$. Recall that $|V(X_i)| \le n_i$. We bound the total update time of the algorithm in the following lemma.

Lemma E.14 *The total update time of the algorithm is at most* $O\left(\frac{n^{1+O(\epsilon)}\Delta^5(\log n)^{O(1/\epsilon^2)}}{\varphi^5}\right)$.

Proof: Fix an index $1 \le i \le r$. We partition the execution of the algorithm into *level-i stages*, where each level-i stage starts when InitializeExpander(i) is called (that is, graph X_i is constructed from scratch), and terminates just before the subsequent call to InitializeExpander(i). Recall that, over the course of a level-i stage, at most $\varphi_i|E(X_i^{(0)})|/(20\Delta_i)$ edges are deleted from the graph $X_i^{(0)}$. 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:

- Constructing expander X_i and its embedding $\mathcal{P}_i$ into graph X_{i-1}, using the algorithm from Lemma E.13; the running time is bounded by:

$$\widetilde{O}\left(\frac{\Delta_{i-1}^4 n_{i-1}}{\varphi_{i-1}^3} + n_i^{1+O(\epsilon)} \cdot (\log n)^{O(1/\epsilon^2)}\right);$$

- Initializing the lists $J_i(e)$ for edges $e \in E(X_{i-1})$: the time to initialize all such lists is bounded by the time needed to compute the embedding $\mathcal{P}_i$.
- Initializing and maintaining the ES-Tree τ_{i-1}: the running time is $\widetilde{O}(|E(X_{i-1})| \cdot D_{i-1}) \leq \widetilde{O}(n_{i-1}\Delta_{i-1}\ell_{i-1}) \leq \widetilde{O}\left(n_{i-1}\Delta_{i-1}^2/\varphi_{i-1}\right)$.
- Running the algorithm from Theorem 2.2 for expander pruning on the expander X_i. Since a single level-i stage may involve the deletion of up to $k = \varphi_i|E(X_i^{(0)})|/(20\Delta_i)$ edges from X_i, the running time is bounded by:

$$\widetilde{O}\left(\frac{\Delta_i^2}{\varphi_i^2} \cdot \frac{\varphi_i|E(X_i^{(0)})|}{20\Delta_i}\right) = \widetilde{O}\left(\frac{n_i\Delta_i^2}{\varphi_i}\right).$$

- The remaining work, executed by `DeleteEdge`$(i-1, e)$, for every edge e that is deleted from graph X_{i-1} (including edges incident to the vertices of the pruned out set S_{i-1}), requires $O(\eta_i)$ time per edge, with total time $O(|E(X_{i-1}^{(0)})| \cdot \eta_i) \leq O\left(n_{i-1}\Delta_{i-1}\eta_i\right) = \widetilde{O}\left(n_{i-1}\Delta_{i-1}^3/\varphi_{i-1}^2\right)$.

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 bounded by:

$$\widetilde{O}\left(\frac{\Delta_{i-1}^4 n_{i-1}}{\varphi_{i-1}^3} + n_i^{1+O(\epsilon)} \cdot (\log n)^{O(1/\epsilon^2)} + \frac{n_i\Delta_i^2}{\varphi_i}\right).$$

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`$(i+1)$ and `Delete`(i, e).

Next, we bound the total number of level-i stages. Consider some index $1 < i' \leq r$, and consider a single level-i' stage. Clearly, the total number of edges that are incident to the pruned-out vertices in $S_{i'}$ is bounded by $|E(X_{i'}^{(0)})| \leq O(n_{i'}\Delta_{i'})$. As the embedding $\mathcal{P}_{i'+1}$ of $X_{i'+1}$ into $X_{i'}$ has congestion at most $\eta_{i'+1} \leq \widetilde{O}(\Delta_{i'}^2/\varphi_{i'}^2)$, this can cause at most $\widetilde{O}\left(n_{i'}\Delta_{i'}^3/\varphi_{i'}^2\right)$ edge-deletions from graph $X_{i'+1}^{(0)}$. As a single level-$(i'+1)$ stage requires $k_{i'+1} \geq \Omega\left(\varphi_{i'+1}|E(X_{i'+1}^{(0)})|/\Delta_{i'+1}\right) = \Omega(\varphi_{i'+1}n_{i'+1})$ edge-deletions from $G_{i'+1}^{(0)}$, the number of level-$(i'+1)$ stages that are contained in a single level-i' stage is bounded by:

$$\widetilde{O}\left(\frac{n_{i'}\Delta_{i'}^3/\varphi_{i'}^2}{n_{i'+1}\varphi_{i'+1}}\right) = \widetilde{O}\left(\frac{n^\epsilon\Delta_{i'}^3}{\varphi_{i'}^2 \cdot \varphi_{i'+1}}\right).$$

Since we only need to support at most $\varphi|E(X)|/(20\Delta)$ edge deletions from the original graph X, there is only a single level-0 stage. Recalling that $\varphi_0 = \varphi$, and for all $1 \leq i \leq r$, $\varphi_i = \hat{\varphi} = 1/(\log n)^{O(1/\epsilon)}$, and that $\Delta_0 = \Delta$, and for all $1 \leq i \leq r$, $\Delta_i = O(\log n)$, we get that for all $1 \leq i \leq r$, the total number of level-i stages is bounded by:

$$O\left(\frac{n^{\epsilon i}(\log n)^{O(i)}\Delta^3}{\hat{\varphi}^{3i} \cdot \varphi^2}\right),$$

while the running time of a single level-i phase, for $i > 1$ is bounded by:

$$\begin{aligned}
&\widetilde{O}\left(\frac{\Delta_{i-1}^4 n_{i-1}}{\varphi_{i-1}^3}+n_i^{1+O(\epsilon)}\cdot(\log n)^{O(1/\epsilon^2)}+\frac{n_i\Delta_i}{\varphi_i}\right)\\
&\le \widetilde{O}\left(\frac{n}{n^{(i-1)\epsilon}\hat{\varphi}^3}+n^{(1-i\epsilon)(1+O(\epsilon))}\cdot(\log n)^{O(1/\epsilon^2)}+\frac{n}{n^{i\epsilon}\hat{\varphi}}\right)\\
&\le \widetilde{O}\left(\frac{n^{1+O(\epsilon)}\cdot(\log n)^{O(1/\epsilon^2)}}{n^{i\epsilon}\hat{\varphi}^3}\right).
\end{aligned}$$

For $i=1$, the running time of a single level-1 phase is bounded by:

$$\widetilde{O}\left(\frac{\Delta^4 n}{\varphi^3}+n^{(1+O(\epsilon))(1-\epsilon)}\cdot(\log n)^{O(1/\epsilon^2)}+\frac{n}{\hat{\varphi}n^{\epsilon}}\right).$$

Therefore, for every $1<i\le r$, the total running time for maintaining level-i data structure is bounded by:

$$\widetilde{O}\left(\frac{n^{1+O(\epsilon)}(\log n)^{O(1/\epsilon^2)}}{n^{i\epsilon}\hat{\varphi}^3}\right)\cdot O\left(\frac{n^{\epsilon i}(\log n)^{O(i)}\Delta^3}{\hat{\varphi}^{3i}\cdot\varphi^2}\right)\le \widetilde{O}\left(\frac{n^{1+O(\epsilon)}(\log n)^{O(1/\epsilon^2)}\Delta^3}{\hat{\varphi}^{3i+3}\cdot\varphi^2}\right)$$

Since $i\le r\le 1/\epsilon$, and $\hat{\varphi}=1/(\log n)^{O(1/\epsilon)}$, this is bounded by: $O\left(\frac{n^{1+O(\epsilon)}\Delta^3(\log n)^{O(1/\epsilon^2)}}{\varphi^2}\right)$.

Lastly, the total running time for maintaining the level-1 data structure is bounded by:

$$\widetilde{O}\left(\frac{\Delta^4 n}{\varphi^3}+\frac{n^{1+O(\epsilon)}\cdot(\log n)^{O(1/\epsilon^2)}}{\hat{\varphi}^2}\right)\cdot\widetilde{O}\left(\frac{n^{\epsilon}\Delta^3}{\hat{\varphi}^3\cdot\varphi^2}\right)\le\widetilde{O}\left(\frac{n^{1+O(\epsilon)}\Delta^7(\log n)^{O(1/\epsilon^2)}}{\varphi^5}\right).$$

Summing this up over all $1\le i\le r$, we get that the total update time of the algorithm is at most $O\left(\frac{n^{1+O(\epsilon)}\Delta^7(\log n)^{O(1/\epsilon^2)}}{\varphi^5}\right)$. □

Responding to expander-short-path-query

Lastly, we provide an algorithm for responding to queries expander-short-path-query. Recall that, given a pair of vertices $x,y\in V(X)\setminus S$, the goal is to return a simple x-y path P in $X\setminus S$ of length at most $O\left(\Delta^2(\log n)^{O(1/\epsilon^2)}/\varphi\right)$, in time $O(|E(P)|)$. We call algorithm $\texttt{Query}(0,x,y)$, that is described in Figure 3, which recursively calls $\texttt{Query}(i,x',y')$ for $i>0$. The idea of the algorithm is simple: we use the ES-Tree τ_0 in graph X_0 in order to compute two paths: one path connecting x to some vertex $x'\in V(X_1)$, and one path connecting y to some vertex $y'\in V(X_1)$, and then recursively call $\texttt{Query}(1,x',y')$ to obtain a short path connecting x' to y' in X_1. We then use the embedding $\mathcal{P}_1$ of X_1 into G_0 in order to convert the resulting path into an x'-y' path in X_0. The final path connecting x to y is obtained by concatenating the resulting three paths.

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

Lemma E.15 *For all $0\le i\le r$, algorithm $\texttt{Query}(i,x,y)$, given a pair x,y of vertices in graph X_i, returns a path connecting x to y in graph X_i of length at most $L_i=(48c^2\log^2 n/\hat{\varphi})^{r-i+1}$ for $i>0$, and of length at most $L_0=(48c^2\log^2 n/\hat{\varphi})^{r+1}\cdot\Delta^2/\varphi$ for $i=0$.*

```
ALGORITHM Query$(i, x, y)$

Assumption: vertices $x, y$ lie in graph $X_i$.

 1. If $i = r$, return the unique $x$-$y$ path in three $\tau_r$, of length at most $D_{r+1}$.

 2. Compute, in tree $\tau_i$, a unique path $Q_{x,x'}$ connecting $x$ to some vertex $x' \in V(X_{i+1})$,
    and a unique path $Q_{y',y}$ connecting $y$ to some vertex $y' \in V(X_{i+1})$. The length of
    each path is bounded by $D_{i+1}$.

 3. If $x' = y'$, set $R_{x',y'} = \emptyset$; otherwise set $R_{x',y'} =$ Query$(i+1, x', y')$.

 4. Let $Q_{x',y'}$ be a path in graph $X_i$ obtained by concatenating, for all edges $e' \in R_{x',y'}$,
    the corresponding path $P(e') \in \mathcal{P}_{i+1}$ from the embedding of $X_{i+1}$ into $X_i$; recall
    that the length of each such path $P(e')$ is bounded by $\ell_{i+1}$.

 5. Return the path $Q_{u,v}$, obtained by concatenating three paths: $Q_{x,x'}, Q_{x',y'}$, and
    $Q_{y',y}$.
```

Figure 3: Algorithm `Query`(i, x, y)

Proof: The proof is by induction on i, where the base case is $i = r$. Recall that, as observed already, one corollary from Lemma E.13, every pair x, y of vertices of X_r have a path of length at most $D_{r+1} = 4\ell_{r+1} \leq 4\hat{\ell} = 4c^2 \log^2 n/\hat{\varphi}$ connecting them in X_r. The path returned by algorithm `Query`(r, x, y) has length at most $2D_{r+1} \leq 8c^2 \log^2 n/\hat{\varphi} \leq L_r$.

Assume now that the claim is true for some value $i + 1$; we now prove it for value i. Consider algorithm `Query`(i, x, y). We are guaranteed that the algorithm computes a path $Q_{x,x'}$ connecting x to some vertex $x' \in V(X_{i+1})$, and a path $Q_{y',y}$ connecting y to some vertex $y' \in V(X_{i+1})$, in graph X_i, where the lengths of both paths are bounded by D_{i+1}. The path $R_{x',y'} =$ `Query`$(i+1, x', y')$ has length at most L_{i+1}, and so the resulting path $Q_{x',y'}$ has length at most $\ell_{i+1} \cdot L_{i+1}$. Therefore, altogether, the final x-y path computed by the algorithm has length at most:

$$2D_{i+1} + \ell_{i+1} \cdot L_{i+1} = 8\ell_{i+1} + L_{i+1}\ell_{i+1}.$$

For $i \geq 1$, $\ell_{i+1} = c^2 \log^2 n/\hat{\varphi}$, while $\Delta_i = c \log n$, and so the length of the path is bounded by:

$$8\ell_{i+1} + L_{i+1}\ell_{i+1} \leq 2L_{i+1} \cdot c^2 \log^2 n/\hat{\varphi} \leq (2c^2 \log^2 n/\hat{\varphi}) \cdot (48c^2 \log^2 n/\hat{\varphi})^{r-i} \leq (48c^2 \log^2 n/\hat{\varphi})^{r-i+1}.$$

For $i = 0$, $\ell_1 = c\Delta \log n/\varphi$ and $\Delta_0 = \Delta$, so the length of the path is bounded by:

$$8c \log n/\varphi + L_1 \cdot c\Delta \log n/\varphi \leq (48c\Delta^2 \log n/\varphi) \cdot (48c^2 \log^2 n/\hat{\varphi})^{r-1} \leq (48c^2 \log^2 n/\hat{\varphi})^r \cdot \Delta^2/\varphi.$$

From the algorithm's description it is immediate to verify that the running time is bounded by $O(|E(Q)|)$, where Q is the returned path. □

Recalling that $r = \lfloor 1/\epsilon \rfloor$ and $\hat{\varphi} = 1/(\log n)^{O(1/\epsilon)}$, we obtain the following immediate corollary.

Corollary E.16 *Given any pair of vertices $x, y \in V(G) \setminus S$, algorithm* `Query`$(0, x, y)$ *returns a (possibly non-simple) x-y path Q in $X \setminus S$, of length at most $O\left(\Delta^2 (\log n)^{O(1/\epsilon^2)}/\varphi\right)$, in time $O(|E(Q)|)$.*

E.2.1 Proof of Lemma E.13

The proof of the lemma is very similar to the proof of Lemma E.1. The algorithm employs the cut-matching game outlined in Appendix E.1. The main difference is that the Matching Player is implemented via the following theorem, that was proved in [CGL$^+$19], and its corollary.

Theorem E.17 *(Theorem 3.2 in [CGL$^+$19], builds on similar result of [CK19]) There is a deterministic algorithm, that, given an n-vertex graph $G = (V, E)$ with maximum vertex degree Δ, and two disjoint subsets A, B of its vertices, with $|A| \le |B|$, and integers $z \ge 0$, $\ell \ge 32\Delta \log n$, computes one of the following:*

- *either a collection $\mathcal{P}'$ of paths, each of which connects a distinct vertex of A to a distinct vertex of B, with $|\mathcal{P}'| \ge |A| - z$, such that the length of every path in $\mathcal{P}'$ is at most ℓ, and the paths in $\mathcal{P}'$ cause congestion at most ℓ^2; or*
- *a cut (X, Y) in G, with $|X|, |Y| \ge z/2$, and $\varphi_G(X, Y) \le 72\Delta \log n/\ell$.*

The running time of the algorithm is $\tilde{O}(\ell^3|E(G)| + \ell^2 n)$.

We obtain the following immediate corollary of Theorem E.17.

Corollary E.18 *There is a deterministic algorithm, that, given an n-vertex graph $X = (V, E)$ with maximum vertex degree Δ, that is a φ-expander, and two disjoint subsets A, B of its vertices, with $|A| \le |B|$, computes a collection $\mathcal{P}'$ of paths, where each path in $\mathcal{P}'$ connects a distinct vertex from A to a distinct vertex from B, and $|\mathcal{P}'| = |A|$, such that the paths in $\mathcal{P}'$ cause congestion at most $O\left(\frac{\Delta^2 \log^2 n}{\varphi^2}\right)$, and every path has length at most $O\left(\frac{\Delta \log n}{\varphi}\right)$. The running time of the algorithm is $\tilde{O}(\Delta^4 n/\varphi^3)$.*

Proof: We set the parameter $\ell = \frac{150\Delta \log n}{\varphi}$, and apply the algorithm from Theorem E.17 to graph X, vertex sets A and B, and the parameter ℓ, setting $z = 0$. Notice that the algorithm may not return a cut (X, Y) with $\varphi_G(X, Y) \le 72\Delta \log n/\ell < \varphi$, because graph X is a φ-expander. Therefore, it must return a collection $\mathcal{P}$ of paths, with each path connecting a distinct vertex from A to a distinct vertex from B, and $|\mathcal{P}| = |A|$. The length of each path in $\mathcal{P}$ is bounded by $\ell \le O\left(\frac{\Delta \log n}{\varphi}\right)$, and the paths in $\mathcal{P}$ cause congestion at most $\ell^2 \le O\left(\frac{\Delta^2 \log^2 n}{\varphi^2}\right)$. The running time of the algorithm is bounded by $\tilde{O}(\ell^3|E(G)| + \ell^2 n) \le \tilde{O}(\Delta^4 n/\varphi^3)$. □

The remainder of the proof of Lemma E.13 is almost identical to the proof of Lemma E.1. We first need to take care of the special case when $\epsilon < c''/\log k$, where c'' is the constant from Corollary E.9. In this case, $k < 2^{c''/\epsilon}$ holds, and, since we have assumed that $\varphi \le 1/2^{\Omega(1/\epsilon)}$, we can assume that $\varphi \le 1/k$. In this case, we let X' be an arbitrary constant-degree φ'-expander, where $\varphi' = \Omega(1)$, with $V(X') = T$. In order to embed this expander into X, we select, for every edge $e = (x, y) \in E(X')$, a shortest x-y path $P(e)$ connecting x to y in X; from Observation 2.1, the length of $P(e)$ is at most $O\left(\frac{\Delta \log n}{\varphi}\right)$. We then let $\mathcal{P} = \{P(e) \mid e \in E(X')\}$ be the embedding of X' into X. Clearly, the congestion of this embedding is bounded by $k \le 1/\varphi$. The running time of this algorithm is bounded by $O(k|E(X)|) \le O(n\Delta/\varphi)$. Therefore, we assume from now on that $\epsilon \ge c''/\log k$.

We start with a graph X', whose vertex set is T, and edge set is empty. We initialize $\mathcal{P} = \emptyset$. For convenience, denote $|T| = k$; assume for now that k is an even integer. We then perform at most $O(\log k)$ iterations. The ith iteration is performed as follows. First, we apply the algorithm from

Corollary E.9 to graph X'. Assume for now that the algorithm produces a partition (A_i, B_i) of $V(X')$ with $|A_i|, |B_i| \geq k/4$, $|A_i| \leq |B_i|$, and $|E_{X'}(A_i, B_i)| \leq k/100$. We let (A'_i, B'_i) be any partition of $V(X')$ with $|A'_i| = |B'_i|$ and $A_i \subseteq A'_i$. Next, we apply the algorithm from Corollary E.18 to graph X' and vertex sets A'_i, B'_i. The algorithm returns a collection $\mathcal{P}_i$ of paths, where each path in $\mathcal{P}_i$ connects a distinct vertex from A'_i to a distinct vertex from B'_i and $|\mathcal{P}_i| = |A'_i|$, such that the paths in $\mathcal{P}'$ cause congestion at most $O\left(\frac{\Delta^2 \log^2 n}{\varphi^2}\right)$, and every path has length at most $O\left(\frac{\Delta \log n}{\varphi}\right)$. We let M_i be the perfect matching between vertices of A'_i and vertices of B'_i, where $(x, y) \in M_i$ iff some path in $\mathcal{P}_i$ connects x to y. We add the paths in $\mathcal{P}_i$ to $\mathcal{P}$, and continue to the next iteration.

Finally, assume that the algorithm from Corollary E.9 returns a subset $S \subsetneq V(X')$ of at least $k/2$ vertices of X', such that graph $X[S]$ is a φ'-expander, for $\varphi' = 1/(\log k)^{O(1/\epsilon)}$. Let i be the index of the current iteration. We then set $A'_i = V(X) \setminus S$ and $B'_i = S$, and apply the algorithm from Theorem E.11 to graph X', and vertex sets A'_i, B'_i. We continue exactly as before, obtaining a matching M_i between the vertices of A'_i and the vertices of B'_i, of cardinality $|A'_i|$, and its corresponding routing $\mathcal{P}_i$. We let M_i be a partial matching between vertices of A'_i and vertices of B'_i, where $(x, y) \in M'_i$ iff some path in $\mathcal{P}_i$ connects x to y, so $|M_i| = |A'_i|$. We then add the edges of M_i to graph X', add the paths in $\mathcal{P}_i$ to set $\mathcal{P}$, and terminate the algorithm, returning the graph X' and its embedding $\mathcal{P}$.

From Theorem E.10, the total number of iterations in the algorithm is bounded by $O(\log k)$, and the final graph X' is a $\hat{\varphi}$-expander, for $\hat{\varphi} = 1/(\log k)^{O(1/\epsilon)}$. The paths in $\mathcal{P}$ provide an embedding of the graph X' into X, where the length of every path in $\mathcal{P}$ is at most $O\left(\frac{\Delta \log n}{\varphi}\right)$. Since the number of iterations is at most $O(\log k)$, and each set $\mathcal{P}_i$ of paths causes edge-congestion at most $O\left(\frac{\Delta^2 \log^2 n}{\varphi^2}\right)$, the total edge-congestion caused by the paths in $\mathcal{P}$ is at most $O\left(\frac{\Delta^2 \log^3 n}{\varphi^2}\right)$.

Recall that we have assumed that $|T|$ is an even integer. If this is not the case, then we let $t \in T$ be any vertex, and we run the same algorithm as above, replacing the set T of terminals with $T' = T \setminus \{t\}$. Let X' be the resulting $\hat{\varphi}$-expander, with $V(X') = T'$, and let $\mathcal{P}$ be its embedding into X. Let $t' \in T'$ ba any vertex, and let P' be the shortest path connecting t to t' in graph X. From Observation 2.1, the length of P' is at most $O(\Delta \log n/\varphi)$. We then obtain the final expander X'' from X', by inserting the vertex t and the edge (t, t') into it. The set $\mathcal{P} \cup \{P'\}$ of paths defines an embedding of X'' into X with edge-congestion bounded by $O\left(\frac{\Delta^2 \log^3 n}{\varphi^2}\right)$, with path-lengths bounded by $O\left(\frac{\Delta \log n}{\varphi}\right)$. It is easy to verify that graph X'' remains a φ''-expander, for $\varphi'' = 1/(\log k)^{O(1/\epsilon)}$. It now remains to analyze the running time of the algorithm.

The algorithm consists of $O(\log k)$ iterations. Each iteration involves applying the algorithm from Corollary E.9, whose running time is $O\left(k^{1+O(\epsilon)} \cdot (\log k)^{O(1/\epsilon^2)}\right)$, and the algorithm from Corollary E.18, whose running time is $\widetilde{O}(\Delta^4 n/\varphi^3)$. Therefore, the total running time of the algorithm is $\widetilde{O}(\Delta^4 n/\varphi^3) + O\left(k^{1+O(\epsilon)} \cdot (\log k)^{O(1/\epsilon^2)}\right)$.

References

[ABCP98] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, 1998.

[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.

[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.

[AP90] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 503–513. IEEE, 1990.

[BBG$^+$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. *arXiv preprint arXiv:2004.08432*, 2020.

[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.

[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.

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

[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.

[BHG$^+$20] Thiago Bergamaschi, Monika Henzinger, Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New techniques and fine-grained hardness for dynamic near-additive spanners. *arXiv preprint arXiv:2010.10134*, 2020.

[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.

[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.

[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.

[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.

[Che18] Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 170–181. IEEE, 2018.

[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 *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 389–400, 2019.

[CS21] Julia Chuzhoy and Thatchaphol Saranurak. Deterministic algorithms for decremental shortest paths via layered core decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2478–2496. SIAM, 2021.

[CZ20] Shiri Chechik and Tianyi Zhang. Dynamic low-stretch spanning trees in subpolynomial time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 463–475. SIAM, 2020.

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

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

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

[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.

[FGH20] Sebastian Forster, Gramoz Goranci, and Monika Henzinger. Dynamic maintenance of low-stretch probabilistic tree embeddings with applications. *CoRR*, abs/2004.10319, 2020.

[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.

[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.

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

[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.

[GVY95] N. Garg, V.V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)-cut theorems and their applications. *SIAM Journal on Computing*, 25:235–251, 1995.

[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.

[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.

[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.

[HKN16] Monika Henzinger, Sebastian Krinninger, 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.

[HKNS15] Monika Henzinger, Sebastian Krinninger, 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 symposium on Theory of computing*, pages 21–30, 2015.

[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.

[KŁ19] Adam Karczmarz and Jakub Łacki. Reliable hubs for partially-dynamic all-pairs shortest paths in directed graphs. *arXiv preprint arXiv:1907.02266*, 2019.

[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.

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

[ŁN20] Jakub Łacki and Yasamin Nazari. Near-optimal decremental approximate multi-source shortest paths. *arXiv preprint arXiv:2009.08416*, 2020.

[LR99] F. T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46:787–832, 1999.

[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.

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

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

[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.

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

[WW18] Virginia Vassilevska Williams and R Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *Journal of the ACM (JACM)*, 65(5):1–38, 2018.

[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.