Decremental All-Pairs Shortest Paths in Deterministic Near-Linear Time

Julia Chuzhoy*
Toyota Technological Institute at Chicago
Chicago, IL, U.S.A.
cjulia@ttic.edu

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 $\tilde{O}(|E(P)|)$, 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 $(\text{poly}\log m)$-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.

CCS CONCEPTS

• **Theory of computation** → **Dynamic graph algorithms**; **Network flows**; **Shortest paths**.

KEYWORDS

Decremental all-pairs shortest paths; minimum multicut.

ACM Reference Format:

Julia Chuzhoy. 2021. Decremental All-Pairs Shortest Paths in Deterministic Near-Linear Time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC '21), June 21–25, 2021, Virtual, Italy.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3406325.3451025

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 dist$'(x, y)$ on

*Supported in part by NSF grant CCF-1616584 and CCF-2006464.

the distance $\mathrm{dist}_G(x,y)$ between x and y in graph G, such that $\mathrm{dist}_G(x,y) \le \mathrm{dist}'(x,y) \le \alpha \cdot \mathrm{dist}_G(x,y)$. 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 $\tilde{O}(|E(P)|)$, 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 [6] 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 $O(kn^{1/k} \log n)$. A recent work of [10] provides a randomized algorithm for maintaining a spanner of a fully dynamic n-vertex graph G with approximation factor $O(\mathrm{poly}\log n)$ 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 $O(n\,\mathrm{poly}\log n)$. One significant advantage of this algorithm over the algorithm of [6] in that, unlike the algorithm of [6], 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 [10] implies a randomized $\mathrm{poly}\log n$-approximation algorithm for APSP that has $O(m\,\mathrm{poly}\log n)$ total update time. A recent work of [7] 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 $\tilde{O}(|E(P)|)$ for shortest-path-query, where P is the path that the query returns, and $O(\mathrm{poly}\log(mL))$ 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 $\tilde{O}(|E(P)|)$ 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 4.

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 [19, 21, 32], 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 [5, 8, 30, 40] is dedicated to this direction. The fastest algorithms in this line of work, due to Henzinger, Krinninger, and Nanongkai [30], and due to Bernstein [8] 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 [20], and Roddity and Zwick [39] showed that, assuming the Boolean Matrix Multiplication (BMM) conjecture[1], for any $\alpha, \beta \ge 1$ with $2\alpha + \beta < 4$, no 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. [31] 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 [43], combined with the work of Roddity and Zwick [39], 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 [1, 13, 25, 30] recently culminated with a randomized algorithm of Chechik [14], that, for any integer $k \ge 1$ 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

[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

is weighted and undirected. This result is near-optimal, as all its parameters almost match the best static algorithm [42]. We note that this result was recently slightly improved by [36], 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 [30], 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 [29]. A recent work of [18] provided a deterministic algorithm for unweighted undirected graphs, that, for any parameter $1 \le k \le o(\log^{1/8} n)$, in response to query shortest-path-query(x,y), returns a path of length at most $3\cdot 2^k \cdot \mathrm{dist}_G(x,y) + 2^{(O(k\log^{3/4} n)}$, 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 [33]).

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 $O(\text{poly}\log m)$-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 Covers*. 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 $\mathrm{dist}_G(v,u)\le D$. 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 $\mathcal{F}$ of vertex-induced subgraphs of G (that we call *clusters*), such that, for every vertex $v\in V(G)$, there is some cluster $C\in\mathcal{F}$ with $B_G(v,D)\subseteq V(C)$. Additionally, we require that for every cluster $C\in\mathcal{F}$, for every pair $x,y\in V(C)$ of its vertices, $\mathrm{dist}_G(x,y)\le\alpha\cdot D$; if this property holds, then we say that $\mathcal{F}$ is a *weak* $(D,\alpha\cdot D)$-neighborhood cover of G. If, additionally, the diameter of every cluster $C\in\mathcal{F}$ is bounded by $\alpha\cdot D$, then we say that $\mathcal{F}$ 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 $\mathcal{F}$. For this static setting of the problem, the work of [3, 4] 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 *partially dynamic* 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 $\mathcal{F}$ of the graph G, and we require that the clusters in $\mathcal{F}$ may only be updated in a specific fashion: once an initial neighborhood cover $\mathcal{F}$ of G is computed, we are only allowed to delete edges or vertices from clusters that lie in $\mathcal{F}$, or to add a new cluster C to $\mathcal{F}$, which must be a subgraph of an existing cluster of $\mathcal{F}$. Additionally, we require that the algorithm supports queries short-path-query(C,v,v'): given two vertices $v,v'\in V$, and a cluster $C\in\mathcal{F}$ 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 $C=\mathrm{CoveringCluster}(v)$ in $\mathcal{F}$, 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 $\mathcal{F}$ 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 dynamic 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 $\mathcal{F}$ 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 $\mathcal{F}$), due to the restrictions that we impose on the types of allowed updates to the clusters of $\mathcal{F}$, 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 [15, 23, 24] 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 at 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 [29, 30] 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 $\mathcal{F}$, there must be a cluster C' containing C that already belongs to $\mathcal{F}$, 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 in 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, that we call *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 $1 \le i \le k$, 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 [22, 28], that can equivalently be viewed as an application of the multiplicative weight update paradigm [2], which essentially reduces the Multicommodity Flow problem to decremental APSP; this reduction was first discovered by [38]. 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 [27, 37], 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 $O(k^{O(1)} \cdot m^{4/3}/\epsilon^{O(1)})$ [34] and $\widetilde{O}(mn/\epsilon^2)$ [38]; 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 [27, 37], 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 [35], which is a central tool in the design of fast algorithms for cut and flow problems (see, e.g. [16]). 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 [18], building on [16], 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

[3]The work of [18] 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 the full version of the paper for completeness.

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. [17, 18, 38]).

In the oblivious-adversary setting, our understanding of the problem is almost complete: a sequence of works [13, 25, 26] 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 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 [20, 39] (that are based on the Boolean Matrix Multiplication conjecture) and of [31] (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 [43], combined with the work of Roddity and Zwick [39], 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 [19, 21, 32], combined with the standard weight rounding technique (e.g. [8, 44]) 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 [9, 11, 12], provided algorithms with total update time $\tilde{O}(n^2\log L)$ and $\tilde{O}(n^{5/4}\sqrt{m}) \le \tilde{O}(mn^{3/4})$, while Gutenberg and Wulff-Nielsen [29] 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 [17, 18], 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 [10] 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 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 $\widetilde{O}(|E(P)|)$ 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 $\ell(e) \ge 1$ 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 $\ell(e) \ge 1$ 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 $D \ge 1$, maintains a weak $(D, \alpha\cdot D)$-neighborhood cover $\mathcal{F}$ of G, for $\alpha = O\left((\log m)^{2^{O(1/\epsilon)}}\right)$, and supports queries* short-path-query(C, v, v')*: given a cluster $C \in \mathcal{F}$, 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 $C =$* CoveringCluster(v) *in $\mathcal{F}$, with $B_G(v, D) \subseteq V(C)$. The algorithm starts with $\mathcal{F} = \{G\}$, and the only allowed changes to the clusters in $\mathcal{F}$ are: (i) delete an edge from a cluster $C \in \mathcal{F}$; (ii) delete an isolated vertex from a cluster $C \in \mathcal{F}$; and (iii) add a new cluster C' to $\mathcal{F}$, where $C' \subseteq C$ for some cluster $C \in \mathcal{F}$. 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 $C \in \mathcal{F}$ to which v ever belonged 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 $\mathcal{F} = \{G\}$. Clearly, this initial cluster set $\mathcal{F}$ 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 $\mathcal{F}$, 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 $\mathcal{M} = \{(s_1,t_1),\dots,(s_k,t_k)\}$ 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 4 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 3.3 for more details and the full version of the paper for a formal proof. We now focus on describing our algorithm for the Sparse 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 $1\le D\le \Theta(m)$. For all $1\le i\le \lceil 1/\epsilon\rceil$, 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 $\mathcal{F}_i$ 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 $C\in\mathcal{F}_i$, 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 $\mathcal{F}$ of the graph G that is possibly maintained recursively: $U=\{u(C)\mid C\in\mathcal{F}\}$. (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 $\mathcal{F}$, where $C\subseteq C'$ for some existing cluster $C'\in\mathcal{F}$. 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 $\mathcal{F}$, 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 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 RecDynNC

[4]A similar approach of recursive composition of emulators was used in numerous algorithms for APSP; see, e.g. [14].

[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., [9, 11, 15, 17].

problem on input graph H, with target distance threshold D, and let $\mathcal{F}$ 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 $C \in \mathcal{F}$, 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 $N(C') \le N(C'')$, $\mathsf{diam}(C') \le O(D \operatorname{poly}\log N)$, 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 $\mathcal{F}$, 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 $\mathcal{F}$ of the input graph H, before it undergoes any updates, by repeatedly incurring this procedure. Later, as the algorithm progresses, and update operations are applied to H, the diameters of some clusters $C \in \mathcal{F}$ 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 $\mathcal{F}$. 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 [37] 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, [15] 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 [37] 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 [37] 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 [3, 4], 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 $\mathcal{F}$. 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 $\mathcal{F}$, 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 $\mathcal{F}'$ of C, then we are guaranteed that for all $C' \in \mathcal{F}'$, $N(C') \le N(C)/\rho$. 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', and an embedding of X into C, via short embedding paths, that causes a low edge-congestion (see the full version of the paper 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 [41] 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 ES-Trees rooted at the endpoints of the edges in E' survives, we can support the short-path-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 $\operatorname{dist}_C(x, y)$ sufficiently large. Once every ES-Tree that we maintain is destroyed, we are guaranteed that all edges of E' are deleted from C. We can then iteratively apply Procedure ProcCut in order to further decompose C into a collection of low-diameter clusters (that is, we compute a collection $\mathcal{F}'$ of subgraphs of C, such that $\mathcal{F}'$ 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 $\mathcal{F}'$ is significantly smaller than C, and contains at most $N(C)/\rho$ regular vertices. We can then initialize the algorithm for solving the MaintainCluster problem on each cluster of $\mathcal{F}'$. 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 ES-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 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 $1 \le i \le \lceil \log D \rceil$, we solve the RecDynNC problem in graph H' with target distance threshold $D_i = 2^i$ recursively. Fix some index $1 \le i \le \lceil \log D \rceil$, and let $\mathcal{F}_i$ 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 $C' \in \mathcal{F}_i$ is significantly smaller that C: namely, $N(C') \le N(C)/\rho \le (N(C))^{1-\epsilon}$. Therefore, we can solve the MaintainCluster problem on each such cluster recursively, and we also do so for every cluster that is later added to $\mathcal{F}_i$. Let $\tilde{\mathcal{F}} = \bigcup_i \mathcal{F}_i$ be the dynamic collection of clusters that we maintain.

We use the set $\tilde{\mathcal{F}}$ 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 $U' = \left\{u(C') \mid C' \in \tilde{\mathcal{F}}\right\}$ 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 $C' \in \mathcal{F}_i$ 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 $\mathcal{F}_i$ evolve, which in turn leads to changes in the contracted graph $\hat{H}$. However, we ensure that all changes to the neighborhood covers $\mathcal{F}_i$ are only of the types allowed by Theorem 1.2, namely: (i) delete an edge from a cluster of $\mathcal{F}_i$; (ii) delete an isolated vertex from a cluster of $\mathcal{F}_i$; or (iii) add a new cluster C'' to $\mathcal{F}_i$, where $C'' \subseteq C'$ for some cluster $C' \in \mathcal{F}_i$. 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 then construct two data structures. First, an ES-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 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 $|E'| \le (N(C))^{1-\epsilon}$, we are guaranteed that $|S| \le (N(C))^{1-\epsilon}$, 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 $\mathcal{F}_i$ of graph $H = C \setminus E'$, for $1 \le i \le \lceil \log D \rceil$. This allows us to maintain the neighborhood covers $\{\mathcal{F}_i\}$, 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 $(\log m)^{\operatorname{poly}(1/\epsilon)}$-approximation algorithm with similar running time guarantees, improving the approximation factor beyond the $(\log m)^{\operatorname{poly}(1/\epsilon)}$ 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 [16]. 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 [9, 11], together with Layered Core Decomposition of [18] 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 the full version of the paper for details).

1.3 Organization

Due to lack of space, most of the proofs are deferred to the full version of the paper. We start with preliminaries in Section 2. In Section 3, we define the Recursive Dynamic Neighborhood Cover problem, and state our main result for it. We also show that the proofs of Theorem 1.1 and Theorem 1.2 follow from this result. Lastly, in Section 4 we provide our algorithm for Maximum Multicommodity Flow and Minimum Multicut, proving Theorem 1.3.

2 PRELIMINARIES

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.

Given a graph G with lengths $\ell(e)$ on edges $e \in E(G)$, for a pair of vertices $u, v \in V(G)$, we denote by $\text{dist}_G(u, v)$ 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 vertex $v \in V(G)$ and a distance parameter $D \geq 0$, we denote by $B_G(v, D) = \{u \in V(G) \mid \text{dist}_G(u, v) \leq D\}$ the *ball of radius D around v*.

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 2.1 (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 $\mathcal{F} = \{C_1, \ldots, C_r\}$ 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, $\text{dist}_G(s, s') \leq D'$.

A set $\mathcal{F}$ 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 $C \in \mathcal{F}$, for every pair $s, s' \in S \cap V(C)$ of vertices, $\text{dist}_C(s, s') \leq D'$.

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

3 VALID INPUT STRUCTURE, VALID UPDATE OPERATIONS, AND THE RECURSIVE DYNAMIC 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 this section, we define such inputs and the types of update operations that we allow for them. We also formally define the Recursive Dynamic Neighborhood Cover problem (RecDynNC) and state our main result for this problem. Lastly, we 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.

Definition 3.1 (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 $\mathcal{F}$ 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 to 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*.

We 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 3.2 (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 total 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 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 $\mathcal{F}$ 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. We will generally ensure that the neighborhood cover $\mathcal{F}$ of G is constructed and maintained in such a way that the total number of clusters of $\mathcal{F}$ 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 initial 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 $O(|E(H^0)|)$.)

3.2 The Recursive Dynamic Neighborhood Cover (RecDynNC) Problem

In this subsection we define the Recursive Dynamic Neighborhood Cover problem. The input to the Recursive Dynamic Neighborhood Cover (RecDynNC) problem is a valid input structure $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E(H)}, D\right)$, 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 $\mathcal{F} = \{H[S] \mid S \in \mathcal{U}\}$ of clusters of H, such that $\mathcal{F}$ 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 $C = \mathsf{CoveringCluster}(v)$ in $\mathcal{F}$, with $B_H(v, D) \subseteq V(C)$;
- for every vertex $x \in V(H)$, a list $\mathsf{ClusterList}(x) \subseteq \mathcal{F}$ of all clusters containing x, and for every edge $e \in E(H)$, a list $\mathsf{ClusterList}(e) \subseteq \mathcal{F}$ 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:

- $\mathsf{DeleteVertex}(S, x)$: given a vertex set $S \in \mathcal{U}$, and a vertex $x \in S$, such that x is an isolated vertex in $H[S]$, delete x from S;
- $\mathsf{AddSuperNode}(S, u)$: 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
- $\mathsf{ClusterSplit}(S, S')$: 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 $\mathcal{F}$ undergo over the course of the algorithm, the corresponding sequence of changes to vertex sets of $\{U(C) \mid C \in \mathcal{F}\}$ must obey the above rules.

In addition to maintaining the above data structures, an algorithm for the RecDynNC problem must support short-path-query(C, v, v') queries: given two **regular** vertices $v, v' \in V$, and a cluster $C \in \mathcal{F}$ 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 RecDynNC problem.

3.3 Statement of Main Technical Result and Proofs of Theorem 1.2 and Theorem 1.1

Our main technical result is a dynamic algorithm for the RecDynNC problem, that is summarized in the following theorem.

Theorem 3.3. *There is a deterministic algorithm for the* RecDynNC *problem, that, on input* $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E(H)}, D\right)$ *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* $\mathcal{F}$ *that the algorithm maintains, to which vertex* v *ever belonged. is bounded by* $W^{O(1/\log\log W)}$*. It also ensures that the neighborhood cover* $\mathcal{F}$ *that it maintains is a strong* $(D, \alpha \cdot D)$*-neighborhood cover for the set* V *of regular vertices of* H*.*

The proof of the theorem is deferred to the full version of the paper due to lack of space. By using recursive composition properties of the RecDynNC problem, we then obtain a deterministic algorithm Alg that has similar properties to those of the algorithm from Theorem 3.3, except that it requires that the dynamic degree bound of the input graph is 2, it achieves approximation factor $\alpha = (\log W)^{2^{O(1/\epsilon)}}$, and total update time $O\left(W^{1+O(\epsilon)} \cdot (\log W)^{O(1/\epsilon^2)}\right)$. Additionally, it only provdies a weak Neighborhood Cover.

Algorithm Alg in turn implies the proof of Theorem 1.2. Indeed, 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 $U = V(G)$. 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 consists 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 $\mathcal{I} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E(H)}, 3D\right)$. 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 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. We run Algorithm Alg for the RecDynNC problem on input $\mathcal{I}$ undergoing the sequence Σ' of update operations. Let $\mathcal{F}$ be the neighborhood cover that the algorithm maintains. We then define a neighborhood cover $\mathcal{F}'$ for graph G as follows. For every cluster $C \in \mathcal{F}$, there is a cluster $C' \in \mathcal{F}'$, which is a subgraph of G induced by vertex set $\{x \in V(G) \mid x' \in V(C)\}$. Recall that cluster set $\mathcal{F}$ is initially defined to be $\mathcal{F} = \{H\}$, so initially, $\mathcal{F}' = \{G\}$ holds. After that, the only changes to vertex sets in $\mathcal{U} = \{U(C) \mid C \in \mathcal{F}\}$ are the allowed changes, that include the following three types of operations: (i) DeleteVertex(R, x): given a vertex set $R \in \mathcal{U}$, and a vertex $x \in R$, such that x is an isolated vertex in $H[R]$, delete x from R; if $x = y'$ for some vertex $y \in V(G)$, then we also delete y from C; (ii) AddSuperNode(R, u): since we do not allow supernode splitting operations, no such updates will be performed; and (iii) 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 $\mathcal{F}'$ 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 $\mathcal{F}$ that contains it. The time that is needed to make all these updates to cluster set $\mathcal{F}'$ is subsumed by the time required to maintain cluster set $\mathcal{F}$.

Consider now some vertex $x \in V(G)$. Recall that the algorithm for the RecDynNC problem maintains a cluster $C \in \mathcal{F}$, with $B_H(x', 3D) \subseteq V(C)$. It is easy to verify that $B_H(x', 3D)$ contains every vertex y' with $y \in B_G(x, D)$. We then set CoveringCluster$(x) = C'$, where $C' \in \mathcal{F}$ is the cluster corresponding to C. Our algorithm can support queries short-path-query(C', x, y) by invoking a query short-path-query(C, x', y') in the data structure maintained by Algorithm Alg; we omit the details and the remainder of the analysis due to lack of space.

Lastly, we note that the proof of Theorem 1.1 easily follows from the proof of Theorem 1.2, by using the standard technique of considering each distance scale $D_i = 2^i$, for $1 \leq i \leq \lceil \log L \rceil$ separately, and using the algorithm from Theorem 1.2 in order to maintain a neighborhood cover for each such distance scale; see the full version of the paper for a formal proof.

4 APPLICATION: FAST ALGORITHM FOR MAXIMUM MULTICOMMODITY FLOW AND MINIMUM MULTICUT

In this section, we provide an algorithm for Minimum Multicut and Maximum Multicommodity Flow, proving Theorem 1.3. Recall that 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 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 standard primal-dual technique-based algorithm of [22, 28] (see also [38]).

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.

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

$$\begin{array}{lll} \text{LP}_2 & & \\ \text{Min} & \sum_{e\in E} x_e & \\ \text{s.t.} & & \\ & \sum_{e\in P} x_e \ge 1 & \forall 1 \le i \le k, P \in \mathcal{P}_i \\ & x_e \ge 0 & \forall e \in E \end{array}$$

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 for edges $e \in E$, where at the beginning, for every edge $e \in E(G)$, we set $x_e = 1/m$. As the algorithm progresses, we may raise 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 $|\mathcal{P}| \neq \emptyset$, 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, $c'_1 \le \text{OPT}_{\text{MCF}} = \text{OPT}_{\text{LP}_1} = \text{OPT}_{\text{LP}_2} \le \alpha c_2$. Therefore, $\text{OPT}_{\text{MCF}} \le \alpha c_2 \le O(\alpha \log m) c'_1$, and $\text{OPT}_{\text{MM}} \ge \text{OPT}_{\text{LP}_2} \ge c'_1 \ge \Omega(c_2/\log m)$. We conclude that we have obtained a solution to the Maximum Multicommodity Flow problem, of value $\Omega(\text{OPT}_{\text{MCF}}/(\alpha \log m))$.

Additionally, we have obtained a fractional solution $\{x'_e\}_{e\in E(G)}$ to LP_2, of value $\alpha c_2 \le O(\alpha \log m)\text{OPT}_{\text{MM}}$. 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 [27, 37]. The resulting deterministic algorithm (that is very similar in nature to our Procedure ProcCut (see the full version of the paper), which in fact was inspired by the algorithm of [27, 37]), 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 $c \le O(\alpha \log m)\text{OPT}_{\text{MM}}$ 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 $\mathcal{O}$.

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 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 $O(|P|)$ in order to process the query, but we will only double the lengths of the edges lying on the path P'. This 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 $j \ge 0$, let $\alpha_j = (\alpha^*)^{2j}$, and let $L_j = m^{j\epsilon}$. We will ensure that the following invariant holds:

(I1) For all $j \ge 0$, 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 $j \ge 0$.

We construct a graph G_j, whose vertex set is $V(G_j) = V(G)$. For every edge $e = (v, v') \in E(G)$, let i_e be the integer such that the length of e in G is $2^{i_e}/m$. For every integer $i_e \le i \le \lceil \log m \rceil$, 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 $\mathcal{F}$ 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 $\ell \le 1/\alpha_{j+1}$ 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 shortest length. Then the length of path P' in graph G_j is bounded by:

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

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 $C = \text{CoveringCluster}(s_i)$ be the cluster of $\mathcal{F}$ containing $B_{G_j}(s, D)$, that the algorithm from Theorem 1.2 maintains. We start by checking, in time $\tilde{O}(1)$, whether $t_i \in C$. If this is not the case, then we are guaranteed that $\text{dist}_G(s_i, t_i) > D = 1/(\alpha^* \cdot \alpha_j)$. 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 short-path-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 $|E(P)| \le 2\alpha^* \cdot L_{j+1}$. 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 short-path-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 short-path-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, $|E(P)| \le 2\alpha^* \cdot m^{\epsilon} |E(P'')|$. 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 short-path-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 $\tilde{O}(1)$ time in the iteration in which the pair is marked as explored. Therefore, the total running time of a single phase is bounded by $\tilde{O}\left(m^{1+O(\epsilon)} (\log m)^{2^{O(1/\epsilon)}} + k\right)$. Since the total number of phases is bounded by $O(1/\epsilon)$, the total running time of the algorithm implementing the oracle is:

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

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

ACKNOWLEDGEMENT

We thank Merav Parter for introducing us to the notion of neighborhood covers and to the work of [3, 4], that was the inspiration for this work.

REFERENCES

[1] 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.
[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] Lisa Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.*, 13(4):505–520, 2000.

[23] 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.

[24] Sebastian Forster, Gramoz Goranci, and Monika Henzinger. Dynamic maintenance of low-stretch probabilistic tree embeddings with applications. *CoRR*, abs/2004.10319, 2020.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Adam Karczmarz and Jakub Łacki. Reliable hubs for partially-dynamic all-pairs shortest paths in directed graphs. *arXiv preprint arXiv:1907.02266*, 2019.

[34] 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.

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

[36] Jakub Łacki and Yasamin Nazari. Near-optimal decremental approximate multi-source shortest paths. *arXiv preprint arXiv:2009.08416*, 2020.

[37] 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.

[38] 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.

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

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

[41] 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.

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

[43] 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.

[44] 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.