A New Deterministic Algorithm for Fully Dynamic All-Pairs Shortest Paths

Julia Chuzhoy
Toyota Technological Institute at Chicago
Chicago, USA
cjulia@ttic.edu

Ruimin Zhang
University of Chicago
Chicago, USA
ruimin@uchicago.edu

ABSTRACT

We study the fully dynamic All-Pairs Shortest Paths (APSP) problem in undirected edge-weighted graphs. Given an n-vertex graph G with non-negative edge lengths, that undergoes an online sequence of edge insertions and deletions, the goal is to support approximate distance queries and shortest-path queries. We provide a deterministic algorithm for this problem, that, for a given precision parameter ϵ, achieves approximation factor $(\log\log n)^{2^{O(1/\epsilon^3)}}$, and has amortized update time $O(n^{\epsilon}\log L)$ per operation, where L is the ratio of longest to shortest edge length. Query time for distance-query is $O(2^{O(1/\epsilon)}\cdot\log n\cdot\log\log L)$, and query time for shortest-path query is $O(|E(P)|+2^{O(1/\epsilon)}\cdot\log n\cdot\log\log L)$, where P is the path that the algorithm returns. To the best of our knowledge, even allowing any $o(n)$-approximation factor, no adaptive-update algorithms with better than $\Theta(m)$ amortized update time and better than $\Theta(n)$ query time were known prior to this work. We also note that our guarantees are stronger than the best current guarantees for APSP in decremental graphs in the adaptive-adversary setting.

In order to obtain these results, we consider an intermediate problem, called Recursive Dynamic Neighborhood Cover (RecDynNC), that was formally introduced in [Chuzhoy, STOC '21]. At a high level, given an undirected edge-weighted graph G undergoing an online sequence of edge deletions, together with a distance parameter D, the goal is to maintain a sparse D-neighborhood cover of G, with some additional technical requirements. Our main technical contribution is twofolds. First, we provide a black-box reduction from APSP in fully dynamic graphs to the RecDynNC problem. Second, we provide a new deterministic algorithm for the RecDynNC problem, that, for a given precision parameter ϵ, achieves approximation factor $(\log\log m)^{2^{O(1/\epsilon^2)}}$, with total update time $O(m^{1+\epsilon})$, where m is the total number of edges ever present in G. This improves the previous algorithm of [Chuzhoy, STOC '21], that achieved approximation factor $(\log m)^{2^{O(1/\epsilon)}}$ with similar total update time. Combining these two results immediately leads to the deterministic algorithm for fully-dynamic APSP with the guarantees stated above.

STOC '23, June 20–23, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9913-5/23/06…$15.00
https://doi.org/10.1145/3564246.3585196

CCS CONCEPTS

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

KEYWORDS

all-pairs shortest path; fully dynamic algorithm.

ACM Reference Format:
Julia Chuzhoy and Ruimin Zhang. 2023. A New Deterministic Algorithm for Fully Dynamic All-Pairs Shortest Paths. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC '23), June 20–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3564246.3585196

1 INTRODUCTION

We study the fully dynamic All-Pairs Shortest-Paths (APSP) problem in weighted undirected graphs. In this problem, the input is an undirected n-vertex graph G with lengths $\ell(e)\geq 1$ on its edges, that undergoes an online sequence of edge insertions and 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 $\text{dist}'(x,y)$ on the distance $\text{dist}_G(x,y)$ between x and y in G, such that $\text{dist}_G(x,y)\leq\text{dist}'(x,y)\leq\alpha\cdot\text{dist}_G(x,y)$. Throughout, we denote $|V(G)|=n$, and we denote by m the total number of edges that are ever present in G; if an edge is deleted from G and then inserted into G multiple times, we count these as different edges. We also denote by Λ the ratio of longest to shortest edge length.

APSP is one of the most fundamental problems in graph algorithms, both in the dynamic and the static settings. Algorithms for this problem often serve as building blocks for designing algorithms for a range of other graph problems and beyond. Interestingly, algorithms for dynamic APSP turned out to be extremely useful in the design of fast algorithms for classical cut, flow, and other graph problems in the static setting. Not surprisingly, this problem has been the subject of extensive study, from many different angles and in various regimes.

A central goal in this area is to obtain algorithms with the strongest possible guarantees for the problem. Specifically, we would like the approximation factor α that the algorithm achieves to be low, and its total update time[1] – the time required to maintain

[1] In the context of fully dynamic algorithms, it is customary to focus on amortized update time per operation, which, in our case, is simply the total update time divided by m. We will use total update time and amortized update time per operation interchangeably, but we will try to clearly distinguish between them to avoid confusion.

its data structures – as close as possible to linear in m. In addition to the approximation factor and the total update time, another important parameter is *query time* – the time it takes to process a single query. Ideally, we would like the query time for dist-query to be $O(\text{poly}\log(n \cdot \Lambda))$, and the query time for shortest-path-query to be close to $O(|E(P)|)$, where P is the path that the algorithm returns, which is close to the best query time we can hope for. Lastly, we distinguish between the *oblivious-adversary* setting, where the sequence of updates to graph G is constructed in advance and may not depend on the algorithm's behavior, and the *adaptive-adversary* setting, where each update to graph G may depend arbitrarily on the algorithm's inner state and past behavior, such as responses to queries. While the oblivious-adversary setting appears significantly easier to handle algorithmically, many applications that rely on algorithms for dynamic APSP require that the algorithm works in the adaptive-adversary setting. It is well known that deterministic algorithms always work against an adaptive adversary. Seeing that the APSP problem itself is used as a building block in many different other setting, designing a deterministic algorithm for the problem is especially desirable.

A straightforward algorithm for the fully-dynamic 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, for any parameter $k \leq O(\log n)$, achieves approximation factor $(2k-1)$, has expected amortized update time $O(k^2 \log^2 n)$ per update operation, and expected spanner size $O(kn^{1+1/k} \log n)$. Unfortunately, this algorithm only works against an oblivious adversary. A recent work of [10] provides a randomized algorithm for maintaining a spanner of a fully dynamic n-vertex graph G that can withstand an adaptive adversary. The algorithm achieves approximation factor $O(\text{poly}\log n)$ and total update time $\widetilde{O}(m)$, and it ensures that the number of edges in the spanner H is always bounded by $O(n \,\text{poly}\log n)$. 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 $\text{poly}\log n$-approximation algorithm for APSP that has $O(m \,\text{poly}\log n)$ 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, and, with current state of the art algorithms, cannot lead to a better than logarithmic approximation.

In this paper, our focus is on developing algorithms for the APSP problem, whose query time is $O(|E(P)| \cdot \text{poly}\log(n \cdot \Lambda))$ for shortest-path-query, where P is the path that the query returns, and $O(\text{poly}\log(n \cdot \Lambda))$ 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, ensuring that query time for shortest-path-query is bounded by $O(|E(P)| \cdot \text{poly}\log(n \cdot \Lambda))$ is often crucial to obtaining fast algorithms for other static graph problems, that use algorithms for APSP as a subroutine.

As mentioned already, there are several parameters of interest that we would like to optimize in algorithms for APSP: namely, query time, total update time, and the approximation factor. Additionally, we would like the algorithm to withstand an adaptive adversary, and ideally to be deterministic. There is a huge body of work that studies the APSP problem, in both the dynamic and the static settings, that tries to optimize or achieve various tradeoffs among these different parameters. Some of this work also only focuses on supporting dist-query queries, and not shortest-path-query. We do not attempt to survey all of this work here, partially because this seems impossible, and partially because it may lead to confusion due to the large number of different settings considered. Instead, we will restrict our attention to the adaptive-adversary setting, where the query time for shortest-path-query is $O(|E(P)| \cdot \text{poly}\log(n \cdot \Lambda))$, where P is the returned path, and query time for dist-query(x, y) is $O(\text{poly}\log(n \cdot \Lambda))$. We will try to survey the most relevant results for this setting, in order to put our results in context with previous work. We will also include some results for APSP in *decremental* graphs, where only edge-deletion updates are allowed.

Low-approximation regime. One major direction of study is to obtain algorithms for APSP whose approximation factor is very close to 1. The classical data structure of Even and Shiloach [23, 25, 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 is dedicated to this direction in the decremental setting [5, 8, 30, 39] and in the fully dynamic setting [14, 22, 40]. In the decremental setting, the fastest algorithms in this line of work, due to [30] and [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 $\log \Lambda$ in the total update time, but can only handle an oblivious adversary. In the fully-dynamic setting, all algorithms cited above have amortized update time per operation at least $\Omega(n^2)$. A very recent result of [12] obtained a $(2+\epsilon)$-approximation for fully-dynamic APSP, with amortized update time $O(m^{1+o(1)})$ per operation. The high running times of the above mentioned algorithms are perhaps not surprising in view of strong lower bounds that are known for the low-approximation setting.

Lower Bounds. A number of lower bounds are known for dynamic APSP with low approximation factor. For example, Dor, Halperin and Zwick [24], and Roddity and Zwick [38] showed

that, assuming the Boolean Matrix Multiplication (BMM) conjecture[2], for any $\alpha, \beta \geq 1$ with $2\alpha + \beta < 4$, no combinatorial algorithm for APSP achieves a multiplicative α and additive β approximation, with total update time $O(n^{3-\delta})$ and query time $O(n^{1-\delta})$ for dist-query, for any constant $0 < \delta < 1$. This result was generalized by [31], who showed the same lower bounds for all algorithms and not just combinatorial ones, assuming the Online Boolean Matrix-Vector Multiplication (OMV) conjecture[3]. The work of Vassilevska Williams and Williams [42], combined with the work of Roddity and Zwick [38], implies that obtaining such an algorithm would lead to subcubic-time algorithms for a number of important static problems on graphs and matrices. A very recent result of [1] provides new lower bounds for the dynamic APSP problem, in the regime where only dist-query queries need to be supported, under either the 3-SUM conjecture or the APSP conjecture. Let $k \geq 4$ be an integer, let $\epsilon, \delta > 0$ be parameters, and let $c = \frac{4}{3-\omega}$ and $d = \frac{2\omega-2}{3-\omega}$, where ω is the exponent of matrix multiplication. Then [1] show that, assuming either the 3-SUM Conjecture or the APSP Conjecture, there is no $(k-\delta)$-approximation algorithm for decremental APSP with total update time $O(m^{1+\frac{1}{ck-d}-\epsilon})$ and query time for dist-query bounded by $O(m^{\frac{1}{ck-d}-\epsilon})$. They also show that there is no $(k-\delta)$-approximation algorithm for fully dynamic APSP that has $O(n^3)$ preprocessing time, and then supports (fully dynamic) updates and dist-query queries in $O(m^{\frac{1}{ck-d}-\epsilon})$ time. Due to these lower bounds, it is natural to focus on somewhat higher approximation factors.

Higher approximation factor. In the regime of higher approximation factors, a long line of work [2, 13, 28, 30] focused on the decremental setting with an oblivious adversary. This direction recently culminated with an algorithm of Chechik [15], that, for any integer $k \geq 1$ and parameter $0 < \epsilon < 1$, obtains a $((2+\epsilon)k-1)$-approximation, with total update time $O(mn^{1/k+o(1)} \cdot \log \Lambda)$, when the input graph is weighted and undirected. This result is near-optimal, as all its parameters almost match the best static algorithm of [41]. This result was recently slightly improved by [36], who obtain total update time $O(mn^{1/k} \cdot \log \Lambda)$, and improve query time for dist-query.

The best currently known results for the fully dynamic setting with an oblivious adversary are significantly weaker. For unweighted graphs, the algorithm of [26] achieves approximation factor $n^{o(1)}$, with amortized update time $n^{1/2+o(1)}$ per operation on unweighted graphs, while the algorithm of [2] achieves a constant approximation factor with expected $o(m)$ amortized update time per operation. In fact the latter paper provides a more general tradeoff between the approximation factor and update time, but in all regimes the expected amortized update time is at least $\Theta(\sqrt{m})$ per operation. Lastly, the the algorithm of [27], based on low-stretch trees, achieves $O(m^{\epsilon})$ update time per operation, with a factor $(\log n)^{O(1/\epsilon)}$-approximation in weighted graphs. All of the above mentioned algorithms for fully-dynamic APSP with oblivious adversary only support dist-query. We are not aware of algorithms that can additionally support shortest-path-query.

In contrast, progress in the adaptive-update setting has been much slower. Until very recently, the fastest algorithm for decremental unweighted graphs [29, 30] only achieved an $\tilde{O}(mn/\epsilon)$ total update time (for approximation factor $(1+\epsilon)$), and the work of [21], for any parameter $1 \leq k \leq o(\log^{1/8} n)$, achieved a multiplicative $3 \cdot 2^k$ and additive $2^{(O(k \log^{3/4} n)}$ approximation, with query time $O(|E(P)| \cdot n^{o(1)})$ for shortest-path-query, and total update time $n^{2.5+2/k+o(1)}$. Until very recently, the fastest adaptive-update algorithms for **weighted** graphs had total update time $O\left(\frac{n^3 \log \Lambda}{\epsilon}\right)$ and approximation factor $(1+\epsilon)$ (see [34]), even in the **decremental** setting.

To summarize, to the best of our knowledge, until very recently, even if we allowed 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 were 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 were known for unweighted undirected graphs, even in the **decremental** setting.

Two very recent results[4] provided significantly stronger algorithms for decremental APSP in weighted graphs: [17] designed a deterministic algorithm, that, for any $\Omega(1/\log\log m) < \epsilon < 1$, achieves approximation factor $(\log m)^{2^{O(1/\epsilon)}}$, and has total update time $O\left(m^{1+O(\epsilon)} \cdot (\log m)^{O(1/\epsilon^2)} \cdot \log \Lambda\right)$. The query time is $O(\log m \log\log \Lambda)$ for dist-query, and $O(|E(P)| + \log m \log\log \Lambda)$ for shortest-path-query, where P is the returned path. The main focus of [12] was mostly on a special case of APSP called Single Source Shortest Paths (SSSP), but they also obtained a deterministic algorithm for decremental APSP with approximation factor $m^{o(1)}$ and total update time $O(m^{1+o(1)})$; unfortunately, the tradeoff between the approximation factor and the total update time is not stated explicitly, though they mention that the approximation factor is super-logarithmic. As mentioned already, they also obtain new results in the low-approximation regime for the fully dynamic setting of APSP: a $(2+\epsilon)$-approximation with amortized update time $O(m^{1+o(1)})$ per operation.

In this paper we improve the results of [17] in two ways. First, we extend the algorithm to the fully-dynamic setting, and second, we improve the approximation factor to $(\log\log m)^{2^{O(1/\epsilon^3)}}$. Altogether, we obtain a deterministic algorithm for fully-dynamic APSP, that, given a precision parameter $\frac{2}{(\log n)^{1/200}} < \epsilon < 1/400$, achieves approximation factor $\alpha = (\log\log n)^{2^{O(1/\epsilon^2)}}$, and has amortized update time $O\left(n^{O(\epsilon)} \cdot \log \Lambda\right)$ per operation (if starting from an empty graph). Query time for dist-query is $O\left(2^{O(1/\epsilon)} \cdot \log n \cdot \log\log \Lambda\right)$, and query time for shortest-path-query is:

$$O\left(|E(P)| + 2^{O(1/\epsilon)} \cdot \log n \cdot \log\log \Lambda\right),$$

where P is the path that the algorithm returns (note that, if we choose $\epsilon \geq 1/\log\log n$, then query time for dist-query becomes

[2]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 $\delta > 0$.

[3]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.

[4]To the best of our knowledge, the two results are independent.

$O(\text{poly}\log(n\cdot\Lambda)$, and query time for shortest-path-query becomes $O(|E(P)| + \text{poly}\log(n\cdot\Lambda))$. An important intermediate problem that we study is Sparse Neighborhood Cover, and its generalization called Recursive Dynamic Neighborhood Cover (RecDynNC) that we discuss next.

Sparse Neighborhood Cover and RecDynNC **problem.** 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 $\text{dist}_G(v, u) \leq D$. Suppose we are given a static graph G with non-negative edge lengths, a distance parameter D, and a desired approximation factor α. A $(D, \alpha \cdot D)$-*neighborhood cover* for G is a collection $\mathcal{C}$ of vertex-induced subgraphs of G (that we call *clusters*), such that, for every vertex $v \in V(G)$, there is some cluster $C \in \mathcal{C}$ with $B_G(v, D) \subseteq V(C)$. Additionally, we require that, for every cluster $C \in \mathcal{C}$, for every pair $x, y \in V(C)$ of its vertices, $\text{dist}_G(x, y) \leq \alpha \cdot D$; if this property holds, then we say that $\mathcal{C}$ is a *weak* $(D, \alpha \cdot D)$-neighborhood cover of G. If, additionally, the diameter of every cluster $C \in \mathcal{C}$ is bounded by $\alpha \cdot D$, then we say that $\mathcal{C}$ is a *strong* $(D, \alpha \cdot D)$-neighborhood cover of G. Ideally, it is also desirable that the neighborhood cover is *sparse*, that is, every edge (or every vertex) of G only lies in a small number of clusters of $\mathcal{C}$. For this static setting of the problem, the work of [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 [17] a new problem, called Recursive Dynamic Neighborhood Cover (RecDynNC) was introduced. The problem can be viewed as an adaptation of Sparse Neighborhood Covers to the dynamic (decremental) setting, but with additional constraints that make it easy to use as a building block in other dynamic algorithms. The input to this problem is a bipartite graph $H = (V, U, E)$, with non-negative lengths $\ell(e)$ on edges $e \in E$, and a distance parameter D. Vertices in set V are called *regular vertices*, while vertices in set U are called *supernodes*. Graph H undergoes an online sequence Σ of updates, each of which must be of one of the following three kinds: (i) edge deletion; or (ii) isolated vertex deletion; or (iii) supernode splitting. In the latter kind of update, we are given a supernode $u \in U$, and a collection $E' \subseteq \delta_H(u)$ of its incident edges. We need to insert a new supernode u' into H, and, for every edge $e = (u, v) \in E'$, insert an edge (u', v) into H. We note that, while, in general, graph H is decremental, the supernode-splitting update allows us to insert edges into it, in a limited fashion. For conciseness, we will refer to an input $\mathcal{J} = (H = (V, U, E), \{\ell(e)\}_{e\in E}, D)$ as described above, as *valid input structure*, and to edge-deletion, isolated vertex-deletion, and supernode-splitting updates as *valid update operations*. Since edges may be inserted into graph H via supernode-splitting updates, in order to control the size of the resulting graph, another parameter called *dynamic degree bound* is used. We say that the dynamic degree bound of valid input structure $\mathcal{J}$ that undergoes a sequence Σ of valid update operations is μ if, for every regular vertex v, the total number of edges that are ever present in H and are incident to v, is bounded by μ.

The goal in the RecDynNC problem is to maintain a weak $(D, \alpha \cdot D)$-neighborhood cover $\mathcal{C}$ of the graph H. However, we require that the clusters in $\mathcal{C}$ are only updated in a specific fashion: once an initial neighborhood cover $\mathcal{C}$ of H is computed, we can only update clusters via *allowed changes*: for each cluster C, we can delete edges or vertices from C, and, additionally, if some supernode $u \in V(C)$ just underwent a supernode-splitting update, we can insert the resulting new supernode u' and all edges connecting it to other vertices of C, into cluster C. A new cluster C' may only be added to $\mathcal{C}$, if there is a cluster $C \in \mathcal{C}$ with $C' \subseteq C$. In this case, we say that cluster C underwent a *cluster-splitting* update. The algorithm must also maintain, for every regular vertex v of H, a cluster $C = \text{CoveringCluster}(v) \in \mathcal{C}$, with $B_H(v, D) \subseteq V(C)$. Additionally, we require that the neighborhood cover is *sparse*, namely, for every regular vertex v of H, the total number of clusters of $\mathcal{C}$ to which v may ever belong over the course of the algorithm is small. Lastly, we require that the algorithm supports queries short-path-query(C, v, v'): given two vertices $v, v' \in V$, and a cluster $C \in \mathcal{C}$ with $v, v' \in C$, return a path P in the current graph H, of length at most $\alpha \cdot D$ connecting v to v' in G, in time $O(|E(P)|)$, where α is the approximation factor of the algorithm.

Given any edge-weighted decremental graph G and a distance bound D, it is easy to transform G into a valid input structure: we simply view the vertices of G as supernodes, and we subdivide its edges with new vertices, that become regular vertices in the resulting bipartite graph H. An algorithm for solving the RecDynNC problem on the resulting valid input structure $\mathcal{J}$ (that only undergoes edge-deletion updates) then naturally allows us to maintain a sparse neighborhood cover in the original graph G. However, the specific definition of the RecDynNC problem makes it more versatile, and more specifically, we can naturally compose instances of the problem recursively with one another.

A typical way to exploit this composability property is the following. Suppose we solve the RecDynNC problem on a bipartite graph H, with some distance bound D. Let $\mathcal{C}$ be the collection of clusters that the resulting algorithm maintains. Assume now that we would like to solve the same problem on graph H, with a larger distance bound $D' > D$. We can then construct another graph H', whose set of regular vertices is the same as that in H, and the set of supernodes is $\{u(C) \mid C \in \mathcal{C}\}$. We add an edge $(v, u(C))$ to the graph if and only if regular vertex v lies in cluster $C \in \mathcal{C}$, and we set the lengths of the resulting edges to be D. As the clusters in $\mathcal{C}$ evolve, we can maintain graph H' via valid update operations: when some cluster $C \in \mathcal{C}$ undergoes cluster-splitting, and a new cluster $C' \subseteq C$ is created, we can apply supernode-splitting to supernode $u(C)$ in order to update graph H' accordingly. It is not hard to verify that the resulting graph H' is an emulator for H, with respect to distances that are greater than D. We can then scale all edge lengths down by factor D, and solve the problem on graph H', with a new, significantly smaller, distance parameter D'/D. If neighborhood cover $\mathcal{C}$ is sparse, and every regular vertex of H ever belongs to at most Δ clusters of $\mathcal{C}$, then the dynamic degree bound for graph H' is bounded by Δ, so graph H' itself is sparse.

We note that, while the RecDynNC problem was first formally defined in [17], the idea of using clustering of a dynamic graph G in order to construct an emulator was exploited before numerous times (see e.g. the constructions of [16, 26, 27] 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. While neither of these algorithms provide neighborhood covers (as can be observed from the fact that one can view the sets of clusters that are maintained for each distance scale as disjoint, something that cannot be achieved in neighborhood covers), a connection between low-diameter decompositions (that often serve as the basis of low-stretch spanning trees) and neighborhood covers has been noticed in prior work. For example, [37], provide a construction of neighborhood covers from low-diameter decompositions. Additionally, all the above-mentioned algorithms are randomized and assume an oblivious adversary. On the other hand, [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 $\mathcal{C}$, there must be a cluster C' containing C that already belongs to $\mathcal{C}$, seems crucial in order to allow an easy recursive composition of emulators obtained from the neighborhood covers, and the requirement that the neighborhood cover is sparse is essential for bounding the sizes of the graphs that arise as the result of such recursive compositions. We also note that a similar approach of recursive composition of emulators was used in numerous algorithms for APSP (see, e.g. [15]), and 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, 16, 20].

It is not hard to verify that an algorithm for the RecDynNC problem immediately implies an algorithm for **decremental** APSP with the same approximation factor, and the same total update time (to within $O(\log \Lambda)$-factor). In [17], a deterministic algorithm for the RecDynNC problem was provided, with approximation factor $\alpha = O\left((\log m)^{2^{O(1/\epsilon)}}\right)$, and total update time:

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

The algorithm ensured that, for every regular vertex $v \in V(H)$, the total number of clusters of $\mathcal{C}$ that v ever belongs to is bounded by $m^{O(1/\log\log m)}$.

In this work, we improve the results of [17] in two ways. First, we provide a black-box reduction from fully dynamic APSP to the RecDynNC problem. Second, we provide an improved algorithm for the RecDynNC problem. The algorithm, given a valid input structure $\mathcal{J} = \left(H, \{\ell(e)\}_{e\in E(H)}, D\right)$ undergoing a sequence of valid update operations, with dynamic degree bound μ, together with parameters $\hat{W}$ and $1/(\log \hat{W})^{1/100} \leq \epsilon < 1/400$, such that, if we denote by N the number of regular vertices in H at the beginning of the algorithm, then $N \cdot \mu = \hat{W}$ holds, achieves approximation factor $\alpha = (\log\log \hat{W})^{2^{O(1/\epsilon^2)}}$, with total update time $O(N^{1+O(\epsilon)} \cdot \mu^{O(1/\epsilon)})$. The algorithm also ensures that, for every regular vertex v, the total number of clusters in the weak neighborhood cover $\mathcal{C}$ that the algorithm maintains, to which v ever belongs over the course of the algorithm, is bounded by $\hat{W}^{4\epsilon^3}$. By combining these two results, we obtain a deterministic algorithm for the fully dynamic APSP problem, that, given a precision parameter $\frac{2}{(\log n)^{1/200}} < \epsilon < 1/400$, achieves approximation factor $\alpha = (\log\log n)^{2^{O(1/\epsilon^2)}}$, and has amortized update time $O\left(n^{O(\epsilon)} \cdot \log \Lambda\right)$ per operation (if starting from an empty graph), with query time $O\left(2^{O(1/\epsilon)} \cdot \log n \cdot \log\log \Lambda\right)$ for dist-query and query time $O\left(|E(P)| + 2^{O(1/\epsilon)} \cdot \log n \cdot \log\log \Lambda\right)$ for shortest-path-query, where P is the path that the algorithm returns. We now state our results more formally, and discuss the techniques that we employ, while pointing out specific remaining bottlenecks for obtaining a better tradeoff between the approximation factor and the update time of the algorithm.

1.1 Our Results

As mentioned already, a problem that plays a central role in this work is RecDynNC. We do not repeat the definition of the problem from above; a formal (and equivalent) definition can be found in Section 3. However, the definition that we provided above omitted one technical detail: the Consistent Covering requirement.

Let $\mathcal{J} = (H = (V, U, E), \{\ell(e)\}_{e\in E}, D)$ be a given valid input structure that undergoes an online sequence Σ of valid update operations. Let $\mathcal{T}$ be the time horizon associated with Σ. In order to define the Consistent Covering property, we first need to define the notion of *ancestor-clusters*. This notion is defined in a natural way. If C is a cluster that is present in $\mathcal{C}$ at the beginning of the algorithm, then for all $\tau \in \mathcal{T}$, $\text{Ancestor}^{(\tau)}(C) = C$, so C is its own ancestor. Assume now that C' is a cluster that was added to set $\mathcal{C}$ at some time $\tau' > 0$, by applying a cluster-splitting update to a cluster $C \in \mathcal{C}$. Then for all $\tau \in \mathcal{T}$, if $\tau < \tau'$, $\text{Ancestor}^{(\tau)}(C') = \text{Ancestor}^{(\tau)}(C)$, and otherwise $\text{Ancestor}^{(\tau)}(C') = C'$.

We are now ready to define the Consistent Covering property. Consider an algorithm for the RecDynNC problem on input $(\mathcal{J}, \Sigma)$, and let $\mathcal{C}$ be the collection of cluster that it maintains. We say that the algorithm obeys the Consistent Covering property, if, for every regular vertex $v \in V(H)$, for every pair $\tau' < \tau \in \mathcal{T}$ of time points, if $C = \text{CoveringCluster}(v)$ at time τ, and $\text{Ancestor}^{(\tau')}(C) = C'$, then, at time τ', $B_H(v, D) \subseteq V(C')$ held. We require that algorithms for the RecDynNC problem obey the Consistent Covering property. Our first result is a reduction from a variant of fully-dynamic APSP to RecDynNC.

1.1.1 *Reduction from Fully-Dynamic* APSP *to* RecDynNC.

We provide a black-box reduction from fully-dynamic APSP to RecDynNC. Our reduction shows that, if there exists an algorithm for the RecDynNC problem with some general set of parameters, then we can convert it into an algorithm for the fully-dynamic APSP problem. The assumption on the existence of an algorithm for RecDynNC, that serves as the starting point of the reduction, is the following.

Assumption 1.1. *There is a deterministic algorithm for* RecDynNC*, that, given a valid input structure $\mathcal{J} = (H = (V, U, E), \{\ell(e)\}_{e\in E}, D)$ undergoing a sequence of valid update operations, with dynamic degree bound μ, together with parameters $\hat{W}$ and $1/(\log \hat{W})^{1/100} \leq$*

$\epsilon < 1/400$, such that, if we denote by N the number of regular vertices in H at the beginning of the algorithm, then $N \cdot \mu \leq \hat{W}$ holds, achieves approximation factor $\alpha(\hat{W})$, with total update time $O(N^{1+O(\epsilon)} \cdot \mu^{O(1/\epsilon)})$. Moreover, the algorithm ensures that, for every regular vertex $v \in V$, the total number of clusters in the weak neighborhood cover $\mathcal{C}$ that the algorithm maintains, to which vertex v ever belongs over the course of the algorithm, is bounded by $\hat{W}^{4\epsilon^3}$. Here, $\alpha(\cdot)$ is a non-decreasing function.

If Assumption 1.1 holds, then it is quite easy to obtain an algorithm for decremental APSP (see Section 3.4.2 in the full version of [17]), that, on an input graph G that initially has m edges, has total update time $O(m^{1+O(\epsilon)+o(1)}(\log m)^{O(1/\epsilon)} \log \Lambda)$, and achieves an approximation factor roughly $\alpha(m)$. One of the main contributions of this work is showing that an algorithm for the RecDynNC problem implies an algorithm for **fully-dynamic** APSP. Specifically, we show that, if Assumption 1.1 holds, then there is an algorithm for a problem that is very similar to, but is slightly different from fully-dynamic APSP. We call this problem D^*-restricted APSP, and define it next. For a dynamic graph G and time τ, we denote by $G^{(\tau)}$ the graph G at time τ.

Definition 1.1 (D^*-restricted APSP *problem).* The input to the D^*-restricted APSP problem is an n-vertex graph G with integral lengths $\ell(e) \geq 1$ on its edges $e \in E(G)$, that undergoes an online sequence Σ of edge deletions and insertions, together with a precision parameter $\frac{1}{(\log n)^{1/200}} < \epsilon < 1/400$, and a distance parameter $D^* > 0$. The goal is to support approximate short-path queries: given a pair $x, y \in V(G)$ of vertices, the algorithm needs to respond "YES" or "NO", in time $O\left(2^{O(1/\epsilon)} \cdot \log n\right)$. If the response is "NO", then $\mathrm{dist}_G(x, y) > D^*$ must hold. If the response is "YES", then the algorithm should be able, additionally, to compute a path P in the current graph G, connecting x to y, of length at most $\alpha' \cdot D^*$, in time $O(|E(P)|)$, where α' is the *approximation factor* of the algorithm.

The following theorem summarizes our reduction from D^*-restricted APSP to RecDynNC.

Theorem 1.2. *Suppose Assumption 1.1 holds. Then there is a deterministic algorithm for the D^*-restricted* APSP *problem, that achieves approximation factor $\alpha' = (\alpha(n^3))^{O(1/\epsilon)}$, and has amortized update time at most $n^{O(\epsilon)}$ per operation, if starting from an empty graph.*

1.1.2 New Algorithm for RecDynNC. Our next result is an improved algorithm for the RecDynNC problem, that is summarized in the following theorem.

Theorem 1.3. *There is a deterministic algorithm for the* RecDynNC *problem, that, given a valid input structure $\mathcal{J} = \left(H, \{\ell(e)\}_{e \in E(H)}, D\right)$ undergoing a sequence of valid update operations, with dynamic degree bound μ, together with parameters $\hat{W}$ and $1/(\log \hat{W})^{1/100} \leq \epsilon < 1/400$, such that, if we denote by N the number of regular vertices in $H^{(0)}$, then $N \cdot \mu \leq \hat{W}$ holds, achieves approximation factor $\alpha = (\log \log \hat{W})^{2^{O(1/\epsilon^2)}}$, with total update time $O(N^{1+O(\epsilon)} \cdot \mu^{O(1/\epsilon)})$. The algorithm ensures that, for every regular vertex $v \in V(H)$, the total number of clusters in the weak neighborhood cover $\mathcal{C}$ that the algorithm maintains, to which vertex v ever belongs over the course of the algorithm, is bounded by $\hat{W}^{4\epsilon^3}$.*

By combining Theorem 1.2 and Theorem 1.3, we immediately obtain the following corollary. We defer its proof to the full version of the paper.

Corollary 1.4. *There is a deterministic algorithm for fully dynamic* APSP, *that, given an n-vertex graph G undergoing an online sequence of edge insertions and deletions, and a precision parameter $\frac{1}{(\log n)^{1/200}} < \epsilon < 1/400$, achieves approximation factor $\alpha' = (\log \log n)^{2^{O(1/\epsilon^2)}}$, and has amortized update time $O\left(n^{O(\epsilon)} \cdot \log \Lambda\right)$ per operation if starting from an empty graph, where Λ is the ratio of longest to shortest edge length. Query time for* dist-query *is $O\left(2^{O(1/\epsilon)} \cdot \log n \cdot \log \log \Lambda\right)$ and for* shortest-path-query *it is $O\left(|E(P)| + 2^{O(1/\epsilon)} \cdot \log n \cdot \log \log \Lambda\right)$, where P is the path that the algorithm returns.*

1.2 Our Techniques

We provide a brief overview of our techniques, starting with the proof of Theorem 1.2.

Reduction from D^*-restricted APSP **to** RecDynNC**.** The description that we provide here is somewhat over-simplified, and is intended for intuition only. We assume that we are given a fully dynamic graph G, that undergoes an online sequence Σ of edge-insertions and deletions, such that $|E(G^{(0)})| + |V(G)| + |\Sigma| = m$, together with a distance parameter D^*, and a precision parameter ϵ. At a high level, we use a rather natural approach. This high-level approach was used before in multiple reductions from fully-dynamic to decremental algorithms (see e.g. [2, 26, 27, 32, 33]), but due to the specific setting of the problem that we consider, the use of this approach in our setting gives rise to a number of new technical challenges that we highlight below. We also provide a brief comparison with previous results where a similar approach was used. Assume for simplicity that $q = 1/\epsilon$ is an integer, and that so is $M = m^{\epsilon}$. Assume further that the distance parameter D^* is an integral power of 2. The data structures that we maintain are partitioned into $q+1$ *levels*. We also define a hierarchical partition of the time horizon $\mathcal{T}$ into phases.

For level $L = 0$, there is a single level-0 phase, that spans the entire time horizon $\mathcal{T}$. We maintain a level-0 graph H^0, that is constructed as follows. Let G' be the dynamic graph that is obtained from the input graph G, by ignoring all edge insertions, and only executing edge-deletion updates. Graph H^0 is a bipartite graph, that has a regular vertex $v(x)$ for every vertex $x \in V(G)$, and a regular vertex $v(e)$ for every edge $e \in E(G')$. Additionally, it has a supernode $u(x)$ for every vertex $x \in V(G)$, that connects, with an edge of length 1, to the corresponding regular vertex $v(x)$. For every edge $e = (x, y) \in E(G')$, we also connect $v(e)$ to $u(x)$ and $u(y)$, with edges of length $\ell(e)$. As graph G' undergoes edge-deletions, the corresponding bipartite graph H^0 undergoes edge-deletions as well. For every integer $0 \leq i \leq \log D^*$, we can view graph H^0 as an instance of the RecDynNC problem, with distance bound $D_i = 2^i$. We apply the algorithm for RecDynNC from Assumption 1.1 to this instance, and we denote by $\mathcal{C}_i^0$ the resulting collection of clusters that it maintains. For every cluster $C \in \mathcal{C}_i^0$, we say that the *scale* of C is i, and we denote $\mathrm{scale}(C) = i$. Let $\mathcal{C}^0 = \bigcup_{i=0}^{\log D^*}$ be the collection of all level-0 clusters.

Consider now some level $0 < L \leq q$. We partition the time horizon $\mathcal{T}$ into at most M^L *level-L phases*. Each level-L phase Φ_k^L spans exactly M^{q-L} consecutive edge-insertion updates from the update sequence Σ for graph G, except for possibly the last phase that may contain fewer edge insertions. We define this hierarchical partition of the time horizon so that, for all $0 < L \leq q$, every level-L phase is contained in some level-$(L-1)$ phase.

Consider now some level $0 < L \leq q$ and a level-L phase Φ_k^L. Let $\Phi_{k'}^{(L-1)}$ be the unique level-$(L-1)$ phase that contains Φ_k^L. We associate, with phase Φ_k^L, a collection A_k^L of edges of G, that the level-L data structure will be "responsible" for during phase Φ_k^L. These are all the edges that were inserted into G since the beginning of level-$(L-1)$ phase $\Phi_{k'}^{(L-1)}$, but before the beginning of level-L phase Φ_k^L. It is easy to see that the number of such edges must be bounded by M^{q-L+1}. We also denote by S_k^L the collection of vertices of G that serve as endpoints of the edges of A_k^L.

We are now guaranteed that, at all times $\tau \in \mathcal{T}$, for every edge $e \in E(G)$, either $e \in E(G^{(0)})$ (in which we say that it lies at level 0); or there is some level $0 < L \leq q$, such that $e \in A_k^L$ currently holds, where k is the index of the current level-L phase (in which case we say that the level of e is L). For every path P in graph G, we also define the *level* of path P to be the largest level of any of its edges.

Consider now some level $0 < L \leq q$, and recall that there are at most M^L level-L phases. At the beginning of every level-L phase, we construct level-L data structures from scratch. These data structures consist of a dynamic level-L bipartite graph H^L, that is viewed as an input to the RecDynNC problem. The set of regular vertices of H^L is S_k^L, where k is the index of the current level-L phase. Intuitively, graph H^L will be "responsible" for all level-L paths in graph G. We describe the sets of supernodes and of edges of H^L below. For all $0 \leq i \leq \log D^*$, we view graph H^L, together with distance parameter $D_i = 2^i$, as an instance of the RecDynNC problem, and we apply the algorithm from Assumption 1.1 to this instance, denoting the resulting collection of clusters by $\mathcal{C}_i^L$. We say that all clusters in $\mathcal{C}_i^L$ have scale i, and we denote $\mathcal{C}^L = \bigcup_{i=0}^{D^*} \mathcal{C}_i^L$. The supernodes of graph H^L are vertices $u(C)$ corresponding to some of the clusters $C \in \bigcup_{L'<L} \mathcal{C}^{L'}$. As the clusters in set $\bigcup_{L'<L} \mathcal{C}^{L'}$ evolve, we maintain the corresponding dynamic graph H^L via valid update operations, where, for example, a cluster-splitting update of a cluster $C \in \bigcup_{L'<L} \mathcal{C}^{L'}$ can be implemented via a supernode-splitting update applied to supernode $u(C)$.

Notice that, while the number of level-L phase may be as large as M^L, the number of regular vertices in the level-L graph H^L is bounded by $2M^{q-L+1}$. Therefore, even though we need to recompute a level-L data structure from scratch at the beginning of each level-L phase, the size of the corresponding graph is sufficiently small that we can afford it. The level-q data structure is computed from scratch after every edge-insertion update, though the number of regular vertices in the corresponding graph H^q is bounded by $M \leq m^\epsilon$.

While the high-level idea described above is quite natural, and was used multiple times in the past (see e.g. [2, 26, 27, 32, 33]), it poses a number of challenges. The main challenge is the coordination between the different levels that is needed in order to support short-path queries. Consider, for example, a short-path query between a pair x, y of vertices of G, and assume that there is a path P in G connecting x to y, whose length is $D < D^*$. Notice, however, that the edges of P may belong to different levels, and there may not be a single level L, such that all vertices of P lie in the graph H^L. Assume that the level of path P is L. Then we would like the level-L data structure to be "responsible" for this query. In other words, we would like some path P', whose length is comparable to D, to represent path P in graph H^L. But it is possible that the endpoints x and y of P do not even lie in graph H^L, so it is not clear which path in H^L we should use as a representative of path P.

This issue seems especially challenging in the setting of APSP with adaptive adversary, where it is required that approximate short-path-query queries are supported. For comparison, [2] and [27] use a very similar high-level idea of a hierarchical partition of the time horizon and the set of edges. In [27], the algorithm is only required to maintain a low-stretch probabilistic tree embedding of the graph. This allows them to combine the trees maintained at different levels into a single tree that has a relatively low height, thereby circumventing the problem of coordinating between graphs from different levels. In order to respond to dist-query between a pair of vertices, they simply compute the length of the path between the two vertices in the tree that they maintain. Their data structure however cannot support approximate shortest-path-query. If we tried to similarly combine graphs from different levels in order to overcome the challenge of coordinating between them, we would obtain another fully dynamic (non-tree) graph, and it is not clear how to support approximate shortest-path-query in this graph. A different approach was taken by [2], whose algorithm exploits specific properties of the distance oracles of [39, 41]. The latter constructions however are randomized and can only withstand an oblivious adversary.

In order to resolve this issue of coordination between levels, we associate, to every cluster $C \in \bigcup_{L=0}^{q} \mathcal{C}^L$, a set $V^F(C) \subseteq V(G)$ of vertices, and we think of cluster C as representing this collection of vertices of G. For a level $0 \leq L \leq q$, we include in graph H^L supernodes $u(C)$ for all clusters $C \in \bigcup_{L'<L} \mathcal{C}^{L'}$ with $S_k^L \cap V^F(C) \neq \emptyset$, where k is the index of the current level-L phase. For every vertex $x \in S_k^L$, and supernode $u(C)$ with $x \in V^F(C)$, we add an edge $(v(x), u(C))$ to graph H^L, whose length is $2^{\text{scale}(C)}$. The main challenge in this construction is to define the sets $V^F(C)$ of vertices for clusters $C \in \bigcup_{L=0}^{q} \mathcal{C}^L$. On the one hand, we would like to make these sets broad enough, so that the resulting graphs H^L are rich enough in order to allow us to support approximate short-path queries. On the other hand, in order to ensure that the algorithm is efficient, these sets cannot be too large.

In order to support short-path queries between pairs of vertices $x, y \in V(G)$, we employ a notion of "covering chains" – structures that span multiple levels. Suppose the shortest path P connecting x to y in G has length $D \leq D^*$, and belongs to level L. Using the covering chains, we compute small collections $R(x), R(y) \subseteq S_k^L$ of vertices associated with x and y respectively, such that there exists a vertex $x' \in R(x)$ and a vertex $y' \in R(y)$, together with a path P' in graph H^L connecting $v(x')$ to $v(y')$, whose length is comparable to D. Conversely, we show that any such path in H^L

can be transformed into a path in graph G that connects x to y, and has length that is not much larger than D.

Next, we provide a high-level overview of the proof of Theorem 1.3. We also point out the main remaining bottlenecks to obtaining a better approximation.

Improved algorithm for RecDynNC**.** The RecDynNC problem can be effectively partitioned into two subproblems. The first subproblem, called MaintainCluster problem, is responsible for maintaining a single cluster. Suppose we are given any such cluster $C \subseteq H$, where H is the current graph, and a distance parameter $D^* > D$. Cluster C will undergo a sequence Σ_C of valid update operations that correspond to the updates applied to H, possibly with some additional edge-deletions and isolated vertex-deletions. The goal of the MaintainCluster problem is to support short-path-query queries: given a pair x, y of regular vertices of C, compute a path P of length at most $\alpha \cdot D^*$ connecting them in graph C, in time $O(|E(P)|)$, where α is the approximation factor that the algorithm achieves. Whenever the diameter of cluster C becomes too large, the algorithm may raise a flag F_C, and to provide a pair x, y of regular vertices of C (that we call a *witness pair*), such that $\mathsf{dist}_C(x, y) > D^*$. After that, the algorithm will receive, as part of the update sequence Σ_C, a sequence of edge-deletions and isolated vertex-deletions (that we call a *flag-lowering sequence*), following which at least one of the vertices x, y is deleted from C, and flag F_C is lowered. If the diameter of C remains too large, the algorithm can raise the flag again immediately. Queries short-path-query may not be asked when flag F_C is up. MaintainCluster problem was defined in [17], and we employ the same definition here.

The second problem is MaintainNC problem. This problem is responsible for managing the neighborhood cover $\mathcal{C}$ itself. Initially, we start with $\mathcal{C}$ containing a single cluster - cluster H. The clusters in $\mathcal{C}$ may only undergo allowed operations that are defined exactly like in the RecDynNC problem. The algorithm also needs to maintain, for every regular vertex $v \in V(H)$, a cluster $\mathsf{CoveringCluster}(v) \in \mathcal{C}$, that contains all vertices of $B_H(v, D)$, so that the Consistent Covering property holds. The algorithm does not need to support any queries. But, at any time, it may receive a cluster C and a pair x, y of vertices of C, such that $\mathsf{dist}_C(x, y) > D^*$ holds (for a parameter D^* that we specify below). It must then produce a flag-lowering sequence Σ' for C (that is, a sequence of edge- and isolated vertex-deletions, after which at least one of x, y is deleted from C). All updates from Σ' must be then applied to cluster C, but they may be interspersed with cluster-splitting operations, when new clusters $C' \subseteq C$ are added to $\mathcal{C}$. The algorithm must also ensure that every regular vertex of H only belongs to a small number of clusters over the course of the time horizon.

By combining the algorithms for the MaintainCluster and the MaintainNC problems, it is easy to obtain an algorithm for the RecDynNC problem, whose approximation factor is $\alpha \cdot D^*/D$, where α is the approximation factor of the algorithm for MaintainCluster, and D^* is the threshold parameter for raising the flags $\{F_C\}_{C \in \mathcal{C}}$.

While [17] did not explicitly define the MaintainNC problem, they effectively provided a simple algorithm for it, that relies on a variation of the standard ball-growing technique, and uses parameter $D^* = \Omega(D \cdot \log N)$, where N is the number of regular vertices in graph H. This overhead of $O(\log N)$ factor is one of the reasons for the $(\log N)^{2^{O(1/\epsilon)}}$-approximation factor that their algorithm achieves, and it is one of the barriers to obtaining a better approximation. We provide a different algorithm for the MaintainNC problem, that allows us to set $D^* = O(D \cdot \log \log N)$. The overhead of factor $O(\log \log N)$ in this algorithm is the only remaining barrier to obtaining an improved algorithm for the RecDynNC problem, and for APSP. For example, if we could ensure that $D^* = O\left(2^{2^{O(1/\mathrm{poly}(\epsilon))}} \cdot D\right)$ is sufficient, we would obtain an algorithm for RecDynNC and for fully dynamic APSP with approximation factor $2^{2^{O(1/\mathrm{poly}(\epsilon))}}$ and the same update time immediately.

In the remainder of this overview, we focus on the MaintainCluster problem. We first provide a brief overview of the algorithm from [17], and then describe our improvements.

The central concept that [17] use in designing an algorithm for the MaintainCluster problem is that of a *balanced pseudocut*, which they also introduced. Let N be the number of regular vertices in $H^{(0)}$, and let μ be the dynamic degree bound. Recall that, as input to the MaintainCluster problem, we are given a cluster C of H, that undergoes a sequence Σ_C of valid update operations with dynamic degree bound μ, and a distance parameter D^*. We use an additional parameter ρ; it may be convenient to think of $\rho = N^{\epsilon}$. Let $\hat{D} > D^*$ be another distance parameter; its specific value is not important for this technical overview, but it is close to D^*. A $(\hat{D}, \rho)$-pseudocut in graph C is a collection T of regular vertices of C, such that, for every regular vertex $v \in V(C) \setminus T$, $B_{C \setminus T}(v, \hat{D})$ contains at most N/ρ regular vertices. This notion can be viewed as a generalization of the balanced vertex multicut, that can be defined as a collection T of vertices, such that every connected component of $C \setminus T$ contains at most N/ρ vertices. Intuitively, once the vertices of the pseudocut (or of a balanced multicut) are deleted from C, we can break it into significantly smaller clusters, while still maintaining the covering properties of the neighborhood cover $\mathcal{C}$. However, balanced pseudocuts have one additional crucial property: [17] provided an algorithm, that, given a $(\hat{D}, \rho)$-pseudocut T in cluster C, either (i) computes an expander graph X, with $V(X) \subseteq T$, such that $|V(X)|$ is comparable to $|T|$, together with an embedding of X into C via short paths that cause a relatively low congestion; or (ii) computes another $(\hat{D}, \rho)$-pseudocut T' in C, with $|T'| \ll |T|$. We denote this algorithm Alg. This algorithm is the core technical part in the algorithm of [17] for the MaintainCluster problem, and our main technical contribution to the MaintainCluster problem essentially replaces algorithm Alg with a different algorithm. We now provide a very brief description of algorithm Alg.

Algorithm Alg**.** A central observation that is needed for the algorithm is the following: let T be a $(\hat{D}, \rho)$-pseudocut in graph C, and suppose we have computed a relatively small subset E' of edges of C, and a collection $T_1, T_2, \ldots, T_{\rho+1}$ of subsets of vertices of T, such that each such subset T_i is sufficiently large, and, for all $1 \leq i < j \leq \rho + 1$, $\mathsf{dist}_{C \setminus E'}(T_i, T_j) > 4\hat{D}$. Then we can compute a pseudocut T' for graph C with $|T'| \ll |T|$. The idea is that there must be some index $1 \leq i \leq \rho + 1$, such that $B_{C \setminus E'}(T_i, 2\hat{D})$ contains at most N/ρ regular vertices. By replacing set T_i in the pseudocut T with the endpoints of the edges in E', we obtain a significantly smaller pseudocut T'. Algorithm Alg starts with the given pseudocut T, and then attempts to compute an expander X over a large subset of vertices of T, and to embed it into C via the Cut-Matching

Game of [35] (in fact, they need to use a weaker variant of the game from [19], who provide a deterministic algorithm for the cut player, but unfortunately only ensure a rather weak expansion in the resulting graph X, which also contributes to the relatively high approximation factor of [17]). If the Cut-Matching Game fails to construct the expander X and embed it into C as required, then it produces two large subsets $T', T'' \subseteq T$ of vertices, and a relatively small subset E' of edges, such that $\mathrm{dist}_{C \setminus E'}(T', T'') > 4\hat{D}$. Then they recursively apply the same algorithm to T' and to T''. After ρ such iterations, if the algorithm failed to construct the desired expander X and its embedding, we obtain large vertex subsets $T_1, \ldots, T_{\rho+1} \subseteq T$, and a subset E' of edges of C, that allow us to compute a much smaller pseudocut, as described above. Even though they perform ρ iterations of the algorithm for the Cut-Matching Game, since, in case of a failure, the subsets $T', T'' \subseteq T$ of vertices that it produces are very large compared to $|T|$, the resulting subsets $T_1, \ldots, T_{\rho+1}$ of vertices are still sufficiently large to make progress. We now complete the description of the algorithm of [17] for the MaintainCluster problem.

The algorithm is partitioned into phases. Initially, we construct a pseudocut T that contains all regular vertices of C. At the beginning of each phase, we use Algorithm Alg (possibly iteratively), in order to compute a pseudocut T', and an expander X defined over a large subset of vertices of T', together with an embedding of X into C via short path that cause a low congestion. Assume first that $|T'| > N^{1-\Theta(\epsilon)}$. The algorithm of [17] employs an algorithm for APSP in expanders on graph X. This algorithm can maintain a large "core" $S \subseteq V(X)$, over the course of a large number of edge-deletions from C (the number that is roughly comparable to $|T'|$). It can also support queries in which, given a pair $x, y \in V(S)$ of vertices, a path of length at most roughly $(\log N)^{O(1/\mathrm{poly}(\epsilon))}$ connecting x to y in X is returned. This path can then be transformed into a path of comparable length connecting x to y in C, using the embedding of X into C. Additionally, they maintain an ES-Tree in graph C, that is rooted at the vertices of S. This tree can be used in order to ensure that all regular vertices of C are sufficiently close to the core S, and, whenever this is not the case, flag F_C is raised. Once the algorithm for APSP in expanders can no longer maintain the core S (after roughly $|T'|$ deletions of edges from C), the phase terminates. It is easy to verify that, as long as the cardinality of the pseudocut T' is sufficiently large (say at least $N^{1-\Theta(\epsilon)}$), the number of phases remains relatively small, and the algorithm can be executed efficiently. Once the cardinality of the pseudocut T' becomes too small, the last phase begins, during which the pseudocut T' remains unchanged. We omit here the description of this phase, since our implementation of this part is essentially identical to that of [17]. We only note that this phase solves the RecDynNC problem recursively on two instances, whose sizes are significantly smaller than that of H. The two instances are then composed in a natural way, which eventually leads to the doubly-exponential dependence of the approximation factor on $1/\mathrm{poly}(\epsilon)$.

The algorithm of [17] for the MaintainCluster problem loses a super-logarithmic in N approximation factor via this approach, that contributes to the final $(\log N)^{2^{1/\mathrm{poly}(\epsilon)}}$-approximation factor for the RecDynNC problem. This loss is largely due to the use of expander graphs. In addition to the issues that we have mentioned with the implementation of the Cut-Matching game via a deterministic algorithm, all currently known algorithms for APSP in expanders only achieve a superlogarithmic approximation factor, and even if they are improved, the loss of at least a polylogarithmic approximation factor seems inevitable. It is typical for this issue to arise when relying on expander graphs for distance-based problems, such as APSP. A recent work of [18] suggested a method to overcome this difficulty, by replacing expander graphs with *well-connected* graphs. Intuitively, if G is a graph, and S is large subset of its vertices, we say that G is well-connected with respect to S (or just well-connected) if, for every pair $A, B \subseteq S$ of disjoint equal-cardinality subsets of vertices of S, there is a collection $\mathcal{P}$ of paths in G, that connects every vertex of A to a distinct vertex of B, such that the paths in $\mathcal{P}$ are short, and they cause a low congestion. In a typical setting, if G is an n-vertex graph, then the lengths of the paths in $\mathcal{P}$ are bounded by $2^{\mathrm{poly}(1/\epsilon)}$, and the congestion that they cause is bounded by $n^{O(\epsilon)}$. [18] also developed a toolkit of algorithmic techniques around well-connected graphs, that mirror those known for expanders. For example, they provide an analogue of the Cut-Matching Game, that, given a graph C and a set T of its vertices, either computes a large set $S \subseteq T$ of vertices, and a graph X with $V(X) \subseteq T$, that is well-connected with respect to S, together with an embedding of X into C via short paths that cause a low congestion; or it computes two relatively large sets $T', T'' \subseteq T$ of vertices, and a small set E' of edges, such that $\mathrm{dist}_{C \setminus E'}(T', T'')$ is large. Additionally, they provide an algorithm for APSP in well-connected graphs, that has similar properties to the above mentioned algorithm for APSP in expanders, but achieves a much better approximation factor of $2^{1/\mathrm{poly}(\epsilon)}$. By replacing expander graphs with well-connected graphs in the algorithm for MaintainCluster problem of [17], we avoid the superlogarithmic loss in the approximation factor that their algorithm incurred. We note however that replacing expanders with well-connected graphs in algorithm Alg is quite challenging technically, for the following reason. Recall that, in the approach that used the Cut-Matching Game, if the algorithm fails to compute an expander X containing a large number of vertices from the given set T and embed it into C, it provides two very large subsets $T', T'' \subseteq T$ of vertices, together with a small set E' of edges, such that $\mathrm{dist}_{C \setminus E'}(T', T'') > 4\hat{D}$. Unfortunately, the analogous algorithm of [18], in case of a failure to embed a well-connected graph, provides vertex sets T', T'', whose cardinalities are significantly smaller than that of T. Specifically, it only ensures that $|T'|, |T''| \geq |T|^{1-4\epsilon^3}/4$. Since we need to continue applying this algorithm recursively, until $\rho + 1$ subsets $T_1, \ldots, T_{\rho+1}$ of vertices of T are constructed, we can no longer guarantee that, for all i, $|T_i|$ is sufficiently large. As a result, if our algorithm fails to compute a well-connected graph X and its embedding into C, we can no longer compute a new pseudocut whose cardinality is significantly lower than that of T. Since the time required to execute this algorithm is super-linear in $|E(C)|$, we cannot afford to execute it many times, so it is critical for us that the cardinality of the pseudocut T decreases significantly with every execution. Our main technical contribution to the algorithm for the MaintainCluster problem is overcoming this hurdle, and designing an analogue of

Algorithm Alg that works with well-connected graphs instead of expanders.

Organization. We start with preliminaries in Section 2. In Section 3, we formally define valid input structure, valid update operations, and the RecDynNC problem. We also provide the statement our main technical result for the RecDynNC problem – an algorithm whose guarantees are somewhat weaker than those in Theorem 1.3, which however allows us to prove Theorem 1.3. Section 4 is dedicated to the reduction from fully dynamic APSP to RecDynNC, and the proof of Theorem 1.2. Due to lack of space, most technical details and formal proofs are deferred to the full version of the paper.

2 PRELIMINARIES

All graphs in this paper are simple, so they may not contain loops or parallel edges. Given a graph G, we say that a graph C is a *cluster* of G, if C is a connected vertex-induced subgraph of G.

Distances, Balls, and Neighborhood Cover. Suppose we are given a graph G with lengths $\ell(e) > 0$ on its edges $e \in E(G)$. For a path P in G, we denote its length by $\ell_G(P) = \sum_{e \in E(P)} \ell(e)$. For a pair of vertices $u, v \in V(G)$, we denote by $\mathrm{dist}_G(u, v)$ the *distance* between u and v in G: the smallest length $\ell_G(P)$ of any path P connecting u to v in G. The *diameter* of the graph G, denoted by $\mathrm{diam}(G)$, is the maximum distance between any pair of vertices in G. Consider now some vertex $v \in V(G)$, and a distance parameter $D \geq 0$. The *ball of radius D around v* is defines as: $B_G(v, D) = \{u \in V(G) \mid \mathrm{dist}_G(u, v) \leq D\}$.

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 the set S of vertices in G is a collection $\mathcal{C} = \{C_1, \ldots, C_r\}$ of clusters of G, 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, $\mathrm{dist}_G(s, s') \leq D'$.

A set $\mathcal{C}$ of clusters of G is a *strong (D, D')-neighborhood cover* for vertex set S if it is a weak (D, D')-neighborhood cover for S, and, additionally, for every cluster $C \in \mathcal{C}$, for every pair $s, s' \in S \cap V(C)$ of vertices, $\mathrm{dist}_C(s, s') \leq D'$. If the set S of vertices is not specified, then we assume that $S = V(G)$.

3 VALID INPUT STRUCTURE, VALID UPDATE OPERATIONS, AND THE RECURSIVE DYNAMIC RECURSIVE NEIGHBORHOOD COVER PROBLEM

Throughout this paper, we will work with inputs that have a specific structure. This structure is identical to the one defined in [17], and it is designed in a way that will allow us to naturally compose different instances recursively, by exploiting the notion of neighborhood covers. In order to avoid repeatedly defining such inputs, we provide a definition here, and then refer to it throughout the paper. We also define the types of update operations that we allow for such inputs. After that, we formally define the Recursive Dynamic Neighborhood Cover problem (RecDynNC). In this section we also state our algorithm for the RecDynNC problem with slightly weaker guarantees, that allows us to prove Theorem 1.3.

3.1 Valid Input Structure and Valid Update Operations

We start by defining a valid input structure; the definition is identical to the one from [17].

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 $\mathcal{J} = \left(H = (V, U, E), \{\ell(e)\}_{e \in E(H)}, D\right)$. If the distance threshold D is not explicitly defined, then we set it to ∞.

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

For brevity of notation, we will refer to edge-deletion, isolated vertex-deletion, and supernode-splitting operations as *valid update operations*. Notice that valid update operations may not create new regular vertices. A supernode splitting operation, however, adds a new supernode to 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, making it challenging to analyze the running times of various algorithms that we run on subgraphs $C \subseteq H$ in terms of $|E(C)|$. In order to overcome this difficulty, we use the notion of the *dynamic degree bound*, which was also defined in [17].

Definition 3.2 (Dynamic Degree Bound). We say that a valid input structure $\mathcal{J} = \left(H, \{\ell(e)\}_{e \in E(H)}, D\right)$, undergoing an online sequence Σ of valid update operations has *dynamic degree bound* μ if, for every regular vertex $v \in V(H)$, the total number of edges incident to v that are ever present in H over the course of the time horizon $\mathcal{T}$ is at most μ.

We will usually denote by $N^0(H)$ the number of regular vertices in the initial graph H. If $(\mathcal{J}, \Sigma)$ have dynamic degree bound μ, then

we are guaranteed that the number of edges that are ever present in H over the course of the update sequence Σ is bounded by $N^0(H) \cdot \mu$.

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

3.2 The Recursive Dynamic Neighborhood Cover (RecDynNC) Problem

In this subsection we provide a formal definition of the Recursive Dynamic Neighborhood Cover problem from [17].

Problem Definition. The input to the Recursive Dynamic Neighborhood Cover (RecDynNC) problem is a valid input structure $\mathcal{J} = (H = (V, U, E), \{\ell(e)\}_{e \in E}, D)$, where graph H undergoes an online 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 is added at the end of the current ordering. The goal is to maintain the following data structures:

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

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

- $\mathsf{DeleteVertex}(S, x)$: given a vertex set $S \in \mathcal{U}$, and a vertex $x \in S$, delete x from S;
- $\mathsf{AddSuperNode}(S, u)$: if u is a supernode that is lying in S, that just underwent a 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 $\mathcal{C}$ undergo over the course of the algorithm, the corresponding sequence of changes to vertex sets in $\{U(C) \mid C \in \mathcal{C}\}$ must obey the above rules.

We note that, while we require that, at the beginning of the algorithm, $\mathcal{U} = \left\{V(H^{(0)})\right\}$ holds, we allow the data structure to update this initial collection of vertex subsets via allowed operations, before processing any updates to graph H. We sometimes refer to the resulting collection $\mathcal{C}$ of clusters, that is obtained before any update from Σ is processed, as *initial collection of clusters*, or *collection of clusters at time* 0.

Ancestor Clusters. It will be convenient for us to define the notion of *ancestors* of clusters in $\mathcal{C}$. Let $\mathcal{T}$ be the time horizon of the update sequence Σ, and let C be a cluster that ever belonged to $\mathcal{C}$ over the course of the algorithm. For every time $\tau \in \mathcal{T}$, we will define an *ancestor* of cluster C at time τ, denoted by $\text{Ancestor}^{(\tau)}(C)$. The definition is inductive over the time when cluster C was first added to $\mathcal{C}$.

Consider first the initial set $\mathcal{C}$ of clusters, that the algorithm constructs prior to processing the first update in Σ. For every cluster $C \in \mathcal{C}$, for every time $\tau \in \mathcal{T}$, we set $\text{Ancestor}^{(\tau)}(C) = C$. Consider now some time $\tau' \in \mathcal{T}$ with $\tau' > 0$, when a new cluster C' is added to set $\mathcal{C}$. Then there is some cluster $C \in \mathcal{C}$, so that cluster C' was split off from cluster C at time τ'. For every time $\tau \in \mathcal{T}$, if $\tau < \tau'$, we set $\text{Ancestor}^{(\tau)}(C') = \text{Ancestor}^{(\tau)}(C)$, and otherwise we set $\text{Ancestor}^{(\tau)}(C') = C'$.

Consistent Covering Property. We require that the data structure for the RecDynNC problem obeys the *Consistent Covering property*, that is defined as follows.

Definition 3.3 (Consistent Covering Property). We say that a data structure for the RecDynNC problem maintains the *Consistent Covering property*, if the following holds. Consider any times $\tau' < \tau$ during the time horizon, and a regular vertex $x \in V(H^{(\tau)})$. Assume that, at time τ, $\mathsf{CoveringCluster}(x) = C$ held, and that $\text{Ancestor}^{(\tau')}(C) = C'$. Then, at time τ', $B_H(x, D) \subseteq V(C')$ held.

The Consistent Covering property was not explicitly defined in [17], but the data structures for the RecDynNC problem provided in that work obey this property. We need this property in order to reduce fully-dynamic APSP to RecDynNC.

In addition to maintaining these data structures, an algorithm for the RecDynNC problem needs to support short-path-query: given two **regular** vertices $v, v' \in V$, and a cluster $C \in \mathcal{C}$ with $v, v' \in C$, return a path P in the current graph H, of length at most $\alpha \cdot D$ connecting v to v' in H, in time $O(|E(P)|)$. This completes the definition of the RecDynNC problem. The *size* of an instance $\mathcal{J} = (H = (V, U, E), \{\ell(e)\}_{e \in E}, D)$ of the RecDynNC instance, that we denote by $N^0(H)$, is the number of regular vertices in the original graph H. In the remainder of the paper, we will always assume that a data structure that an algorithm for the RecDynNC problem maintains must obey the Consistent Covering property.

3.3 Main Technical Result for the RecDynNC Problem and Proof of Theorem 1.3

As one of our main technical results, we prove the following theorem; the proof is deferred to the full version of the paper.

Theorem 3.4. *There is a deterministic algorithm for the* RecDynNC *problem, that, given a valid input structure* $\mathcal{J} = \left(H, \{\ell(e)\}_{e\in E(H)}, D\right)$ *undergoing a sequence of valid update operations, with dynamic degree bound* μ*, together with parameters* $\hat{W}$ *and* $1/(\log \hat{W})^{1/100} \leq \epsilon < 1/400$*, such that, if we denote by* $N^0(H)$ *the number of regular vertices in* H *at the beginning of the algorithm, then* $N^0(H)\cdot\mu \leq \hat{W}$ *holds, achieves approximation factor* $\alpha = (\log\log \hat{W})^{2^{O(1/\epsilon^2)}}$*, with total update time* $O((N^0(H))^{1+O(\epsilon)}\cdot \mu^{O(1/\epsilon)}\cdot D^3)$*. Moreover, the algorithm ensures that for every regular vertex* $v\in V$*, the total number of clusters in the weak neighborhood cover* $\mathcal{C}$ *that the algorithm maintains, to which vertex* v *ever belongs over the course of the algorithm, is bounded by* $\hat{W}^{4\epsilon^4}$*.*

Note that the guarantees provided by Theorem 3.4 are somewhat weaker than those required by Theorem 1.3, in that the total update time of the algorithm depends polynomially on D. We can remove this polynomial dependence on D using standard techniques; a similar idea was used in [17]. We provide the proof of Theorem 1.3 from Theorem 3.4 in the full version of the paper.

4 FROM RecDynNC TO FULLY DYNAMIC APSP

This section is dedicated to the proof of Theorem 1.2. We provide a high level description of the proof here. A detailed proof is deferred to the full version of the paper. We assume that Assumption 1.1 holds, and we denote the algorithm for RecDynNC problem from Assumption 1.1 by $\mathcal{A}$.

We assume that we are given an instance of the D^*-restricted APSP problem, that consists of an n-vertex graph G with integral lengths $\ell(e)\geq 1$ on its edges $e\in E(G)$, together with a precision parameter $\frac{1}{(\log n)^{1/200}} < \epsilon < 1/400$, and a distance parameter $D^* > 0$, where graph G undergoes an online sequence of edge insertions and deletions. For convenience, we denote $\alpha = \alpha(n^3)$, where $\alpha(\cdot)$ is the approximation factor from Assumption 1.1.

As we show in the full version of the paper, by using standard transformations, we can assume that $|V(G)|+|E(G^{(0)})|+|\Sigma|\leq 4m$, where m is the initial number of edges in graph G.

Throughout, we use the parameters $q = \lceil 1/\epsilon\rceil$ and $M = \lceil m^{\epsilon}\rceil$. Notice that $m\leq M^q\leq m^{1+2\epsilon}$. We also use a parameter $\hat{D} = D^*\cdot 2^{10q+10} = D^*\cdot 2^{O(1/\epsilon)}$. For all $0\leq i\leq \log\hat{D}$, we define a distance scale $D_i = 2^i$.

Let $\mathcal{T}$ be the time horizon associated with the update sequence Σ. Consider now graph G at some time $\tau\in\mathcal{T}$, that we denote by $G^{(\tau)}$, and let $e\in E(G^{(\tau)})$ be any edge. We say that an edge $e\in E(G)$ is *original*, if it was present in G at the beginning of the algorithm, and was never deleted or inserted. If edge e is not an original edge, then we say that it is an *inserted* edge.

The data structure that our algorithm maintains is partitioned into $(q+1)$ levels. In order to describe the purpose of each level, we first need to define a partition of the time horizon into phases.

Hierarchical Partition the Time Horizon into Phases. For every level $0\leq L\leq q$, we define a partition of the time horizon $\mathcal{T}$ into *level-L phases*. There is a single level-0 phase, that spans the whole time horizon $\mathcal{T}$. For all $0 < L\leq q$, we partition the time horizon into at most M^L level-L phases, each of which spans a consecutive sequence $\Sigma'\subseteq\Sigma$ of updates, that contains exactly M^{q-L} edge insertions (except for the last phase, that may contain fewer insertions). In other words, if the kth level-L phase ends at time τ, then the τth update operation in Σ is edge-insertion, and, since the beginning of the current level-L phase, exactly M^{q-L} edges have been inserted into G via sequence Σ. It will be convenient for us to ensure that the number of level-L phases is exactly M^L. If this is not the case, then we add empty phases at the end of the last phase.

For $1\leq k\leq M^L$, we denote the kth level-L phase by Φ_k^L, and the subsequence of Σ containing all update operations that occur during Phase Φ_k^L by Σ_k^L. We also associate the time interval $\mathcal{T}_k^L$, corresponding to the update sequence Σ_k^L, with the level-L phase Φ_k^L. For all $0\leq L\leq q$, we will initialize the level-L data structure from scratch at the beginning of each level-L phase. Note that each level-q phase only spans a single edge insertion. In other words, every time a new edge is inserted into G, we start a new level-q phase, and recompute the level-q data structure from scratch. Notice that our definition of phases ensures that, for all $0\leq L' < L\leq q$, every level-L phase is completely contained in some level-L' phase. Intuitively, for all $1\leq L\leq q$, during each level-L phase Φ_k^L, the level-L data structure will be "responsible" for all edges that were inserted into G before the beginning of Phase Φ_k^L, but after the beginning of the current level-$(L-1)$ phase. We now formalize this intuition.

Edge and Path Classification. Consider a level $0 < L\leq q$, and some level-L phase Φ_k^L. Let $\Phi_{k'}^{L-1}$ be the unique level-$(L-1)$ phase that contains Phase Φ_k^L. Let $\tau\in\mathcal{T}$ be the beginning of Phase Φ_k^L, and let $\tau'\in\mathcal{T}$ be the beginning of Phase $\Phi_{k'}^{L-1}$ (note that it is possible that $\tau=\tau'$). We define the set A_k^L of edges of graph G that is associated with Phase Φ_k^L. An edge e belongs to set A_k^L if and only if it was inserted into G between time τ' and time τ (including time τ' and excluding time τ). Notice that the cardinality of set A_k^L is bounded by the number of edges that may be inserted into G during a single level-$(L-1)$ phase, so $|A_k^L|\leq M^{q-L+1}$. The set A_k^L of edges does not change over the course of Phase Φ_k^L. We also denote by S_k^L the collection of vertices of G that serve as endpoints to the edges of A_k^L. Intuitively, level-L data structure is responsible for keeping track of the edges in set A_k^L, over the course of each level-L phase Φ_k^L. We will construct and maintain a level-L graph H^L, that is initialized from scratch at the beginning of each level-L phase Φ_k^L, whose set of regular vertices contains a vertex representing every edge in A_k^L, and a vertex representing every vertex in S_k^L. Observe that, as the level L increases, the cardinalities of the corresponding sets A_k^L of edges decrease, so the graphs that we maintain are smaller. At the same time, as L grows, the number of level-L phases also grows. We will ensure that the time that is required to maintain a level-L data structure over a course of each level-L phase Φ_k^L is almost linear in $|A_k^L|$, allowing us to bound the total update time of the data structure maintained at each level by a function that is almost linear in m. For cosistency of notation, we let Φ_1^0 denote the single level-0 phase, we let A_1^0 be the set of all edges that belonged to G at the beginning of the algorithm.

Consider again some time $\tau \in \mathcal{T}$. For all $0 \leq L \leq q$, we let k_L be the integer, such that $\tau \in \mathcal{T}^L_{k_L}$ holds. We partition all edges of the current graph $G^{(\tau)}$ into $q+1$ *levels*. For $0 \leq L \leq q$, edge e belongs to level L, if and only if $e \in A^L_{k_L}$. It is easy to see that, for every edge e that lies in graph G at time τ, there is precisely one level in $\{0, \ldots, q\}$, to which edge e belongs. We denote the level of edge e by $\text{Level}(e)$. Note that, as the algorithm progresses, the level of a given edge may only decrease.

Consider again graph G at time τ, and let P be any path that is contained in $G^{(\tau)}$, with $|E(P)| \geq 1$. The *level* of path P, denoted by $\text{Level}(P)$, is the largest level of any of its edges, $\text{Level}(P) = \max_{e \in E(P)} \{\text{Level}(e)\}$.

For all $0 \leq L \leq q$, the purpose of the level-L data structure is to support short-path queries between pairs of vertices $x, y \in V(G)$, such that there exists a level-L path in the current graph G connecting x to y, whose length is at most D^*. Since every path connecting x to y in G belongs to one of the levels in $\{0, \ldots, q\}$, this will allow us to support short-path queries as required from the definition of D^*-restricted APSP.

High-Level Description of the Construction. Consider some level $0 \leq L \leq q$, and some level-L phase Φ^L_k. As noted already, at the beginning of Phase Φ^L_k, we initialize the level-L data structure from scratch. Let $\tau \in \mathcal{T}$ denote the time when Phase Φ^L_k begins. Note that τ may also be a starting time of phases from other levels. In such cases, we assume that, when we execute the algorithm for initializing the level-L data structure, then for all $0 \leq L' < L$, the level-L' data structure is already initialized.

Over the course of the level-L phase Φ^L_k, we will maintain a dynamic graph H^L. We will also initialize the corresponding valid input structure $\mathcal{J}^L$, associated with graph H^L, that will undergo a sequence of valid update operations. The set of regular vertices of graph H^L consists of two subsets: set $\left\{v^L(x) \mid x \in S^L_k\right\}$ of vertices, that represent the endpoints of the edges of A^L_k, and set $\left\{v^L(e) \mid e \in A^L_k\right\}$ of vertices, representing the edges of A^L_k. We refer to the former as *type-1 regular vertices* and to the latter as *type-2 regular vertices*. We describe the collection of supernodes of H^L later.

For all $0 \leq i \leq \log \hat{D}$, we will define and maintain a subgraph H^L_i, which is identical to H^L, but it excludes all edges whose length is above D_i. We will also define the corresponding valid input structure $\mathcal{J}^L_i$. We will view $\mathcal{J}^L_i$ as the input to the RecDynNC problem, with distance scale D_i, and we will apply Algorithm $\mathcal{A}$ from Assumption 1.1 to it. We denote by $\mathcal{C}^L_i$ the collection of clusters that this algorithm maintains. For every cluster $C \in \mathcal{C}^L_i$, we say that the *scale* of cluster C is i, and we denote $\text{scale}(C) = i$. We also denote $\mathcal{C}^L = \bigcup_{i=0}^{\log \hat{D}} \mathcal{C}^L_i$ and $\mathcal{C}^{<L} = \bigcup_{L'<L} \mathcal{C}^{L'}$.

We now provide additional details on the structure of the graph H^L, and specifically its supernodes and its edges. The collection of the supernodes of H^L consists of two subsets. The first subset contains, for every vertex $x \in S^L_k$, the corresponding supernode $u^L(x)$, that connects, with an edge of length 1, to the type-1 regular vertex $v^L(x)$. Additionally, for every edge $e \in A^L_k$, such that x is an endpoint of e, we add an edge $(v^L(e), u^L(x))$ of length $\ell(e)$ to graph H^L. We refer to all supernodes we have defined so far as *type-1 supernodes*. The second set of supernodes, called *type-2 supernodes*, contains, for **some** clusters $C \in \mathcal{C}^{<L}$, the corresponding supernode $u^L(C)$.

In order to decide which clusters of $\mathcal{C}^{<L}$ have the corresponding supernode included in graph H^L, and in order to define the edges that are incident to such supernodes, we will define, for every cluster $C \in \mathcal{C}^{<q}$, a decremental set $V^F(C)$ of vertices of G, which we call a *flattened set of vertices*. The specific definition of this set of vertices is somewhat technical and is deferred for later. For a cluster $C \in \mathcal{C}^{<L}$, we add a supernode $u^L(C)$ to graph H^L if and only if $V^F(C)$ contains at least one vertex of S^L_k. If supernode $u^L(C)$ is included in graph H^L, then we connect it with an edge to every type-1 regular vertex $v^L(x)$, for which $x \in V^F(C)$ holds. The length of the edge is $2^{\text{scale}(C)}$. We now proceed to provide intuition on the flattened sets of vertices.

Flattened Sets of Vertices. Consider some level $0 \leq L \leq q$, and some cluster $C \in \mathcal{C}^L$. Intuitively, our layered constructions has created a hierarchical containment structures for the clusters: if, for some cluster $C' \in \mathcal{C}^{<L}$, the correspoinding supernode $u^L(C')$ belongs to cluster C, then we can think of cluster C as "containing" cluster C', in some sense. A natural and intuitive way to define the flattened sets $V^F(C)$ of vertices, would then be the following.

If $C \in \mathcal{C}^0$ is a cluster from level 0, then we let $V^F(C)$ contain every vertex $x \in V(G)$, whose corresponding type-1 regular vertex $v^0(x)$ lies in C. Consider now some level $0 < L \leq q$, and let $C \in \mathcal{C}^L$ be any cluster. As before, for every vertex $x \in V(G)$ with $v^L(x) \in V(C)$, we add vertex x to set $V^F(C)$. But additionally, for every supernode $u^L(C')$ that belongs to cluster C, we add all vertices of $V^F(C')$ to set $V^F(C)$, provided that $\text{scale}(C') \leq \text{scale}(C)$.

This simple intuitive definition of the flattened sets of vertices would serve our purpose in the sense that it would allow us to support the short-path queries as required. But unfortunately, due to the specifics of how the RecDynNC data structure is defined, we cannot control the cardinalities of the resulting flattened sets $V^F(C)$ of vertices, which could in turn lead to a running time that is too high.

In order to overcome this difficulty, we slightly modify the above definition of the flattened set of vertices. Specifically, for every level $0 \leq L \leq q$, and every cluster $C \in \mathcal{C}^L$, we will mark every supernode $u^L(C') \in V(C)$ as either *important* or *unimportant* for cluster C. We only include the vertices of $V^F(C')$ in set $V^F(C)$ if supernode $u^L(C')$ is marked as important for C. A status of a supernode $u^L(C')$ with respect to a cluster C may switch from important to unimportant over the course of the algorithm, but it may never switch in the opposite direction. This allows us to guarantee that the set $V^F(C)$ of vertices remains decremental, which is crucial since the RecDynNC data structure does not support edge insertions, except in the case of supernode splitting. We defer the specific definition of important supernodes to the full version of the paper, but they are defined so that, on the one hand, we can control the cardinalities of the sets $V^F(C) \cap S^L_k$ of vertices (which is sufficient in order to make our construction efficient), while, on the other hand, still allowing us to support short-path queries.

Due to lack of space, the remainder of the proof of Theorem 1.2 is deferred to the full version of the paper.

ACKNOWLEDGMENTS

The first author was supported in part by NSF grants CCF-1616584 and CCF-2006464.

REFERENCES

[1] Amir Abboud, Karl Bringmann, Seri Khoury, and Or Zamir. 2022. Hardness of Approximation in P via Short Cycle Removal: Cycle Detection, Distance Oracles, and Beyond. *arXiv preprint arXiv:2204.10465* (2022).

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

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

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

[5] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. 2007. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *J. Algorithms* 62, 2 (2007), 74–92. https://doi.org/10.1016/j.jalgor.2004.08.004

[6] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. 2012. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms* 8, 4 (2012), 35:1–35:51. https://doi.org/10.1145/2344422.2344425

[7] Thiago Bergamaschi, Monika Henzinger, Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. 2020. New Techniques and Fine-Grained Hardness for Dynamic Near-Additive Spanners. *arXiv preprint arXiv:2010.10134* (2020).

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

[9] Aaron Bernstein. 2017. Deterministic Partially Dynamic Single Source Shortest Paths in Weighted Graphs. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 80. Schloss Dagstuhl-Leibniz-Center for Computer Science.

[10] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. 2020. Fully-dynamic graph sparsifiers against an adaptive adversary. *arXiv preprint arXiv:2004.08432* (2020).

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

[12] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. 2022. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1000–1008.

[13] Aaron Bernstein and Liam Roditty. 2011. 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*. 1355–1365.

[14] Jan van den Brand, Sebastian Forster, and Yasamin Nazari. 2021. Fast Deterministic Fully Dynamic Distance Approximation. *arXiv preprint arXiv:2111.03361* (2021).

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

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

[17] 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*. 626–639. Full version at arXiv:2109.05621.

[18] Julia Chuzhoy. 2022. A Distanced Matching Game, Decremental APSP in Expanders, and Faster Deterministic Algorithms for Graph Cut Problems. SODA 2023, to appear. Full version available at https://home.ttic.edu/~cjulia/papers/APSP-expanders.pdf and on arxiv.

[19] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. 2019. A Deterministic Algorithm for Balanced Cut with Applications to Dynamic Connectivity, Flows, and Beyond. *CoRR* abs/1910.08025 (2019). arXiv:1910.08025 http://arxiv.org/abs/1910.08025

[20] Julia Chuzhoy and Sanjeev Khanna. 2019. 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*. 389–400.

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

[22] Camil Demetrescu and Giuseppe F Italiano. 2004. A new approach to dynamic all pairs shortest paths. *Journal of the ACM (JACM)* 51, 6 (2004), 968–992.

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

[24] Dorit Dor, Shay Halperin, and Uri Zwick. 2000. All-Pairs Almost Shortest Paths. *SIAM J. Comput.* 29, 5 (2000), 1740–1759. https://doi.org/10.1137/S0097539797327908

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

[26] Sebastian Forster and Gramoz Goranci. 2019. 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*. 377–388. https://doi.org/10.1145/3313276.3316381

[27] Sebastian Forster, Gramoz Goranci, and Monika Henzinger. 2020. Dynamic Maintenance of Low-Stretch Probabilistic Tree Embeddings with Applications. *CoRR* abs/2004.10319 (2020). arXiv:2004.10319 https://arxiv.org/abs/2004.10319

[28] Sebastian Forster, Monika Henzinger, and Danupon Nanongkai. 2014. 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*. 146–155.

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

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

[31] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 21–30.

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

[33] Monika R Henzinger and Valerie King. 2001. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Comput.* 31, 2 (2001), 364–374.

[34] Adam Karczmarz and Jakub Łacki. 2019. Reliable Hubs for Partially-Dynamic All-Pairs Shortest Paths in Directed Graphs. *arXiv preprint arXiv:1907.02266* (2019).

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

[36] Jakub Łacki and Yasamin Nazari. 2020. Near-Optimal Decremental Approximate Multi-Source Shortest Paths. *arXiv preprint arXiv:2009.08416* (2020).

[37] Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. 2015. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. 192–201.

[38] Liam Roditty and Uri Zwick. 2011. On Dynamic Shortest Paths Problems. *Algorithmica* 61, 2 (2011), 389–401. https://doi.org/10.1007/s00453-010-9401-5

[39] Liam Roditty and Uri Zwick. 2012. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.* 41, 3 (2012), 670–683.

[40] Mikkel Thorup. 2004. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Scandinavian Workshop on Algorithm Theory*. Springer, 384–396.

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

[42] Virginia Vassilevska Williams and R Ryan Williams. 2018. Subcubic equivalences between path, matrix, and triangle problems. *Journal of the ACM (JACM)* 65, 5 (2018), 1–38.

Received 2022-11-07; accepted 2023-02-06