Bootstrapping Dynamic APSP via Sparsification

Rasmus Kyng
ETH Zurich, Switzerland

Simon Meierhans
ETH Zurich, Switzerland

Gernot Zöcklein
ETH Zurich, Switzerland

Abstract

We give a simple algorithm for the dynamic approximate All-Pairs Shortest Paths (APSP) problem. Given a graph $G = (V, E, \boldsymbol{l})$ with polynomially bounded edge lengths, our data structure processes $|E|$ edge insertions and deletions in total time $|E|^{1+o(1)}$ and provides query access to $|E|^{o(1)}$-approximate distances in time $\widetilde{O}(1)$ per query.

We produce a data structure that mimics Thorup-Zwick distance oracles [17], but is dynamic and deterministic. Our algorithm selects a small number of pivot vertices. Then, for every other vertex, it reduces distance computation to maintaining distances to a small neighborhood around that vertex and to the nearest pivot. We maintain distances between pivots efficiently by representing them in a smaller graph and recursing. We maintain these smaller graphs by (a) reducing vertex count using the dynamic distance-preserving core graphs of Kyng-Meierhans-Probst Gutenberg [16] in a black-box manner and (b) reducing edge-count using a dynamic spanner akin to Chen-Kyng-Liu-Meierhans-Probst Gutenberg [6]. Our dynamic spanner internally uses an APSP data structure. Choosing a large enough size reduction factor in the first step allows us to simultaneously bootstrap a spanner and a dynamic APSP data structure. Notably, our approach does not need expander graphs, an otherwise ubiquitous tool in derandomization.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms; Theory of computation → Sparsification and spanners

Keywords and phrases Dynamic Graph Algorithms, Spanners, Vertex Sparsification, Bootstrapping

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.113

Related Version *Full Version*: `https://arxiv.org/abs/2408.11375`

Funding *Rasmus Kyng*: The research leading to these results has received funding from grant no. 200021 204787 and from the starting grant "A New Paradigm for Flow and Cut Algorithms" (no. TMSGI2_218022) of the Swiss National Science Foundation.
Simon Meierhans: The research leading to these results has received funding from grant no. 200021 204787 of the Swiss National Science Foundation. Simon Meierhans is supported by a Google PhD Fellowship.

1 Introduction

The dynamic *All-Pairs Shortest Paths* (APSP) problem asks to maintain query access to pairwise distances for a dynamic graph $G = (V, E)$. Recently, the setting of maintaining crude distance estimates at low computational cost has received significant attention [15, 5, 11, 9, 8, 2, 12, 13] culminating in deterministic algorithms with sub-polynomial update and query time for graphs receiving both edge insertions and deletions [10, 16, 14]. In contrast to other deterministic approaches, [16] allows efficient *implicit* access to approximate shortest paths via a small collection of dynamic trees. This is crucial for applications to decremental single source shortest paths and incremental minimum cost flows [6].

All existing dynamic approximate APSP data structures that work against adaptive adversaries are highly complex [10, 16, 14].

The algorithm [16] establishes the following *Thorup-Zwick-distance oracle*-inspired framework: To maintain approximate APSP in the graph, we select a small (dynamic) set of pivot vertices. For every other vertex, we can reduce distance computation to (a) computing distances in a small neighborhood around the vertex and (b) computing the distance to the nearest pivot vertex. Then, [16] maintains a smaller graph where distances between the pivot vertices can be computed, and recursively maintains these with the same approach. To maintain this smaller graph, [16] designs a dynamic vertex sparsification procedure and performs edge sparsification using the dynamic spanner of [7, 4]. This spanner internally relies on complex data structures for decremental APSP on expanders [9] and uses expander embedding techniques in a non-trivial way. We will use the overall dynamic Thorup-Zwick approach of [16], and we will maintain a small graph for computing distances between pivots in a simpler way.

A key ingredient for us is a new, more powerful fully-dynamic spanner from [6]. This spanner requires both dynamic APSP and expander embedding techniques to maintain. In this work, we observe that this spanner can be simplified so that it only requires dynamic APSP to maintain. Thus, given a dynamic APSP data structure, we can maintain a fully-dynamic spanner. But, [16] also showed that given a fully-dynamic spanner, we can obtain dynamic APSP. This presents a striking opportunity: we can simultaneously bootstrap a dynamic APSP data structure and spanner. This gives our main result, a dynamic approximate APSP data structure.

▶ **Theorem 1.** *For a graph $G = (V, E, \boldsymbol{l})$ with polynomially bounded integral edge lengths $\boldsymbol{l} \in \mathbb{R}^E_{>0}$ undergoing up to m edge updates (insertions and deletions), there is an algorithm with total update time $m^{1+o(1)}$ that maintains query access to approximate distances $\widetilde{\text{dist}}(u, v)$ such that*

$$\text{dist}(u, v) \leq \widetilde{\text{dist}}(u, v) \leq m^{o(1)} \cdot \text{dist}(u, v).$$

The query time is $\widetilde{O}(1)$.

While we present our theorem in the setting of processing m updates in time $m^{1+o(1)}$ to simplify the presentation, we believe that this algorithm can be extended directly to the worst-case update time setting of [16]. Furthermore, the witness paths can be implicitly represented as forest paths as in [16], which enables the cheap flow maintenance via dynamic tree data structures necessary for applications.

1.1 Roadmap

We first define the necessary preliminaries in Section 2. Then, we present a simplified version of our algorithm as a warm-up in Section 3. Afterwards, we present our bootstrapped algorithm based on the KMG-vertex sparsifier [16] and a dynamic spanner in Section 4.

2 Preliminaries

2.1 Graph Notation

We use $G = (V, E, \boldsymbol{l})$ to refer to a graph G with vertex set V, edge set E and length vector $\boldsymbol{l} \in \mathbb{R}^E_{\geq 1}$. We will restrict our attention to polynomially bounded integral lengths, and will denote their upper bound by L.

2.2 Paths, Distances and Balls

For two vertices u, v, we let $\pi^G_{u,v}$ denote the shortest path from u to v in G (lowest sum of edge lengths). Furthermore, we let $\text{dist}_G(u, v)$ be the length of said path.

We let $B_G(u, v) \stackrel{\text{def}}{=} \{w : \text{dist}(u, w) < \text{dist}(u, v)\}$ and $\overline{B}_G(u, v) \stackrel{\text{def}}{=} \{w : \text{dist}(u, w) \leq \text{dist}(u, v)\}$. Then, for a set $A \subseteq V$ we let $B_G(u, A) \stackrel{\text{def}}{=} \bigcup_{a\in A} B_G(u, a)$ and $\overline{B}_G(u, A) \stackrel{\text{def}}{=} \bigcup_{a\in A} \overline{B}_G(u, a)$ respectively.

Finally, we define the neighborhood of a vertex u as $\mathcal{N}(u) \stackrel{\text{def}}{=} \{v : (u, v) \in E\} \cup \{u\}$ and of a set $A \subseteq V$ as $\mathcal{N}(A) \stackrel{\text{def}}{=} \bigcup_{a\in A} \mathcal{N}(a)$. For a vertex v, we let $E_G(v) \stackrel{\text{def}}{=} \{(v, u) : u \in \mathcal{N}(v)\} \subseteq E(G)$.

When it is clear from the context, we sometimes omit the subscript/superscript G.

▶ **Definition 2** (Spanner). *Given a graph $G = (V, E, \boldsymbol{l})$, we say that a subgraph $H \subseteq G$ is an α-spanner of G if for any $u, v \in V$ we have $\text{dist}_G(u, v) \leq \text{dist}_H(u, v) \leq \alpha \cdot \text{dist}_G(u, v)$.*

2.3 Dynamic Graphs

Vertex splits are a crucial operation in our algorithms. They arise since we contract graph components and represent them as vertices. When such a component needs to be divided, this naturally corresponds to a vertex split defined as follows.

▶ **Definition 3** (Vertex Split). *For a graph $G = (V, E, \boldsymbol{l})$, a vertex split takes a vertex $v \in V$ and a set $U \subseteq \mathcal{N}(v)$, and replaces the vertex v with two vertices v', v'' such that $\mathcal{N}(v') = U$ and $\mathcal{N}(v'') = \mathcal{N}(v) \setminus U$.*

▶ **Definition 4** (Master Nodes). *Initially, each $v \in V(G^{(0)})$ is associated with a unique master node $\text{master}(v) = \{v\}$. Then, at any time t, if some $v \in V(G^{(t)})$ is split into v' and v'', we set $\text{master}(v') = \text{master}(v'') = \{v\} \cup \text{master}(v)$.*

We can also define our notion of fully-dynamic graphs, which includes vertex splits as an operation.

▶ **Definition 5** (Fully-Dynamic Graph). *A fully-dynamic graph is a graph undergoing edge insertions/deletions, vertex splits and isolated vertex insertions.*

Whenever we disallow vertex splits, we refer to edge-dynamic graphs instead.

▶ **Definition 6** (Edge-Dynamic Graph). *An edge-dynamic graph is a graph undergoing edge insertions/deletions and isolated vertex insertions.*

2.4 Dynamic Objects and Recourse

For a dynamic object $(F(t))_{t\in[T]}$ defined on an index set $[T] = \{1, \ldots, T\}$, we refer to $F^{(T')} \stackrel{\text{def}}{=} (F(t))_{t\in[T']}$ as the full sequence up to time T'. We can then apply functions $g(\cdot)$ that operate on sequences to $F^{(T')}$, e.g. $g(F^{(T')})$. This is for example very useful to keep track of how an object changes over time. By providing a difference function $\triangle(F(t), F(t-1))$ for a dynamic object F, one may define the recourse as the sum of differences $\left\|F^{(t)}\right\|_{\rightarrow} \stackrel{\text{def}}{=} \sum_{i=0}^{t-1} \triangle(F(i+1), F(i))$.

For functions f that operate on a static object G instead of on a sequence, we overload the notation by saying $f(G^{(t)})$ is simply f applied to the last object in the sequence, that is, $f(G^{(t)}) = f(G(t))$. For example, $\deg_{\max}(G^{(t)})$ is defined to be the maximum degree of the graph G at time t.

Whenever we make a statements about a dynamic object F without making reference to a specific time t, we implicitly mean that the statement holds at all times. For example, $\|H\|_{\to} \leq \|G\|_{\to}$ formally means for all $t : \left\|H^{(t)}\right\|_{\to} \leq \left\|G^{(t)}\right\|_{\to}$. Meanwhile, $|E(H)| \leq |E(G)|$ formally means for all $t : |E(H^{(t)})| \leq |E(G^{(t)})|$, and by our convention for applying functions of static objects to a dynamic object, we have that $|E(H^{(t)})| \leq |E(G^{(t)})|$ means $|E(H(t))| \leq |E(G(t))|$.

The time scales we use are fine-grained as we will consider multiple dynamic objects that change more frequently than the input graph G does. We will choose an index set for the dynamic sequence which is fine-grained enough to capture changes to all the objects relevant to a given proof.

A dynamic object H might not change in a time span in which another object F changes multiple times. In this case, to have sequences that are indexed by the same set, we simply pad the sequence of H with the same object, so that both dynamic objects would share the same index set T.

For fully-dynamic graphs G, we let $\triangle(G(t), G(t-1))$ denote the number of edge insertions, edge deletions, vertex splits and isolated vertex insertions that happened between $G(t)$ and $G(t-1)$. Note that generally, vertex splits cannot be implemented in constant time, and thus recourse bounds are not directly relevant to bounding running time. Nevertheless, this convention for measuring recourse is useful, as it is tailored precisely to the type of recourse bounds we prove.

We can store anything that occurs at various times throughout our algorithm in a dynamic object and define what recourse means for it. For example, we will repeatedly call a procedure ReduceDegree(X). If at time t we call the function with input $X(t)$, we might want to track the quantity $\sum_{t' \leq t} |X(t')|$, so we simply let R_X be the dynamic object such that $R_X(t) = \sum_{t' \leq t} |X(t')|$. By defining the difference $\triangle(R_X(t), R_X(t-1)) = R_X(t) - R_X(t-1)$, we can find out how much the value changed during two points in time. A statement such as $\|H\|_{\to} \leq \|G\|_{\to} + \|R_X\|_{\to}$ then means that at any time t (suitably fine-grained), the total amount of changes that H underwent are not more than the amount of G, plus some extra number of updates determined by the calls to ReduceDegree$(\cdot)$ made up to that time.

By viewing the run-time of dynamic algorithm as evolving over a sequence of updates and thinking of it as a dynamic object, we also give meaning to the statement, say, that an algorithm runs in time $n + \gamma \|G\|_{\to}$.

2.5 Dynamic APSP

We introduce APSP data structures for edge-dynamic graphs.

▶ **Definition 7** (Dynamic APSP). *For an edge-dynamic graph $G = (V, E, \boldsymbol{l})$, a γ-approximate β-query APSP data structure supports the following operations.*

- AddEdge(u, v) / RemoveEdge(u, v)*: Add/Remove edge (u, v) from/to G.*
- Dist(u, v)*: Returns in time β a distance estimate $\widetilde{\text{dist}}(u, v)$ such that at all times*

$$\text{dist}_G(u, v) \leq \widetilde{\text{dist}}(u, v) \leq \gamma \cdot \text{dist}_G(u, v).$$

- Path(u, v)*: Returns a uv-path P in G of length $|P| \leq$ Dist(u, v) in time $\beta \cdot |P|$.*

Given an APSP data structure, we let $APSP(|E|, \Delta, u)$ denote the total run time of initialization and u calls to AddEdge and RemoveEdge on an edge-dynamic graph with $\deg_{\max}(G) \leq \Delta$ and $|E|$ initial edges.

We sometimes also call γ the (worst-case) stretch of the data structure.

2.6 Degree Reduction Through Binary Search Trees

Given an *edge-dynamic* graph G, we can maintain in time $\widetilde{O}(1) \cdot \|G\|_{\to}$ another edge-dynamic graph G' in which we replace vertices by Binary Search Trees with one leaf node per edge of the vertex in the original graph. It is immediate that if we can find shortest paths in G', we can translate them into shortest paths in G. The advantage of this technique is that while $|E(G')| = O(|E(G)|)$, $|V(G')| = O(|E(G)|)$ and $\|G'\|_{\to} = \widetilde{O}(\|G\|_{\to})$, we have $\deg_{\max}(G') \leq 3$.

3 Warm up: Low-Recourse Thorup-Zwick'05

Before we describe our algorithm, we turn our attention to a much more modest goal that still showcases the main ideas used in our full construction: maintaining a relaxed version of Thorup-Zwick pivots [17] in a fully-dynamic graph with low recourse. This allows us to do away with the careful controlling of maximum degrees in intermediate graphs, which is the root cause of most of the technicalities when making the construction computationally efficient. Furthermore, we do not need to bootstrap our data structure in this setting.

3.1 Pivot Hierarchies

We first introduce pivot hierarchies.

▶ **Definition 8** (Pivot Hierarchy). *For a graph $G = (V, E, \boldsymbol{l})$, we let a pivot hierarchy of depth Λ and cluster size γ be*

- *a collection of vertex sets $\mathcal{C} = A_0, \ldots, A_\Lambda$ with $A_i \subseteq V$ for all i, $A_0 = V$ and $|A_\Lambda| = 1$,*
- *pivot maps $p_i : A_i \to A_{i+1}$ for all $0 \leq i < \Lambda$ and*
- *cluster maps $C_i : A_i \to 2^{A_i}$ for all $0 \leq i < \Lambda$ such that for all $u \in A_i$ $|C_i(u)| \leq \gamma$ and $u \in C_i(u)$.*

Pivot maps should be thought of as mapping vertices to smaller and smaller sets of representatives, and the cluster maps should be thought of as small neighborhoods of pivots for which (approximate) distances are stored. In classical Thorup-Zwick, the cluster $C_i(v)$ consists of all vertices in A_i that are closer to v than $p_i(v)$. In our algorithms, these balls become somewhat distorted due to sparsification. We then define the approximate distances maintained by the hierarchy.

▶ **Definition 9** (Distances). *For a pivot hierarchy of depth Λ, we define $\widetilde{\operatorname{dist}}^{(\Lambda)}(u, v) \stackrel{\text{def}}{=} 0$ for $u, v \in A_\Lambda$ and*

$$\widetilde{\operatorname{dist}}^{(i)}(u, v) \stackrel{\text{def}}{=} \begin{cases} \operatorname{dist}_G(u, v) & \text{if } u \in C_i(v) \\ \operatorname{dist}_G(u, p_i(u)) + \widetilde{\operatorname{dist}}^{(i+1)}(p_i(u), p_i(v)) + \operatorname{dist}_G(p_i(v), v) & \text{otherwise} \end{cases}.$$

We let $\widetilde{\operatorname{dist}}(u, v) \stackrel{\text{def}}{=} \widetilde{\operatorname{dist}}^{(0)}(u, v)$ for $u, v \in V$ and call a pivot hierarchy α distance preserving if $\widetilde{\operatorname{dist}}(u, v) \leq \alpha \cdot \operatorname{dist}_G(u, v)$ for all $u, v \in V$.

We note that efficient distance queries can be implemented as long as the distances within clusters and the distances to pivots are efficiently maintained. In this warm up, we only focus on maintaining the collection $\mathcal{C}$ with low recourse. However, our full algorithm will maintain this additional information.

Formally, the goal of this warm-up section is to prove the following theorem.

▶ **Theorem 10.** *For an edge-dynamic graph $G = (V, E)$ with $n = |V|$ vertices, a $\sqrt{\log n}$-depth $n^{o(1)}$-cluster size pivot hierarchy can be maintained in polynomial time per update with total recourse $\|\mathcal{C}\|_{\to} \leq n^{o(1)} \cdot \|G\|_{\to}$ on the pivot sets.*

We emphasize that this theorem is not particularly useful, as it comes without meaningful guarantees on the update time, and one could just use Dijkstras algorithm to obtain exact distances in this regime. Even the recourse guarantee is also of limited use, as we do not control the recourse of the pivot maps or the cluster maps of the pivot hierarchy. Nevertheless, our algorithm for proving this theorem is instructive and closely mirrors our final dynamic APSP algorithm.

The illustrative algorithm for maintaining pivot hierarchies repeatedly decreases the vertex count via low-recourse core graphs, and the edge count via a low-recourse spanner. We present these two pieces separately.

3.2 Low-Recourse KMG-vertex sparsifier

In this section, we state a theorem about maintaining a KMG-vertex sparsifier with low recourse [16]. Although their algorithm only works for edge-dynamic graphs, it can be extended to fully-dynamic graphs rather straightforwardly in the recourse setting. Recall that we count a vertex split as a single operation when defining the recourse of a fully-dynamic graph.

▶ **Definition 11** (Vertex Sparsifier)**.** *Given $G = (V, E, \boldsymbol{l})$ and $A \subseteq V$, we call a graph $\widetilde{G}$ with vertex set $V(\widetilde{G}) \supseteq A$ a β-approximate vertex sparsifier of G with respect to the set A if for every two vertices $u, v \in V(\widetilde{G})$ we have $\operatorname{dist}_G(u, v) \leq \operatorname{dist}_{\widetilde{G}}(u, v)$ and if $u, v \in A$ we further have $\operatorname{dist}_{\widetilde{G}}(u, v) \leq \beta \cdot \operatorname{dist}_G(u, v)$.*

▶ Remark 12. We always associate a map from $V(\widetilde{G})$ to V with a vertex sparsifier, where multiple vertices in $V(\widetilde{G})$ might map to the same vertex in V. This clarifies what we mean with $\operatorname{dist}_G(u, v) \leq \operatorname{dist}_{\widetilde{G}}(u, v)$ and $\operatorname{dist}_{\widetilde{G}}(u, v) \leq \beta \cdot \operatorname{dist}_G(u, v)$ in Definition 11. When we apply multiple vertex sparsifiers to a graph G, we refer to recursively applying this map as mapping vertices back to G.

▶ **Theorem 13** (See Theorem 3.1 in [16])**.** *Consider a size reduction parameter $k > 1$ and a fully-dynamic graph G with lengths in $[1, L]$. Then, there is a deterministic algorithm that explicitly maintains,*

1. *a monotonically growing pivot set $A \subseteq V(G)$ and pivot function $p : V \to A$, such that $|A| \leq n/k + 2\|G\|_{\to}$ and $|C(u)| \leq \widetilde{O}(k)$ for all $u \in V$ where $C(u) = \{v \in V : \operatorname{dist}_G(u, v) < \operatorname{dist}_G(u, p(u))\}$.*
2. *a fully-dynamic graph $\widetilde{G}$, such that $\widetilde{G}$ is a γ_{VS}-approximate vertex sparsifier of G with respect to A. We have that the initial graph $\widetilde{G}^{(0)}$ has at most $m \cdot \gamma_{VS}$ edges and $\gamma_{VS} \cdot m/k$ vertices, where $\gamma_{VS} = \widetilde{O}(1)$. The following properties hold:*
   a. *$\widetilde{G}$ has lengths $\boldsymbol{l}_{\widetilde{G}}$ in $[1, nL]$, and*
   b. *given any edge $e = (u, v) \in \widetilde{G}$, there exists a uv-path P in G with $l_G(P) \leq l_{\widetilde{G}}(e)$.*
   c. *$\left\|\widetilde{G}\right\|_{\to} \leq \gamma_{VS} \cdot (\|G\|_{\to} + m)$ and $\left\|D_{\widetilde{G}}\right\|_{\to} \leq \gamma_{VS} \cdot \|G\|_{\to}$ where $D_{\widetilde{G}}$ is the dynamic object containing the decremental operations (deletions and vertex splits) on $\widetilde{G}$.*

The algorithm runs in polynomial time.

3.3 Low-Recourse Dynamic Spanner

In this section, we describe a simple algorithm for maintaining a spanner H of a fully-dynamic graph G with remarkably low recourse. The construction is based on the greedy spanner of [1]. The low-recourse property of a variant of this algorithm has been observed in [3].

▶ **Theorem 14** (Dynamic Spanner). *There exists a data structure that given a fully-dynamic graph $G = (V, E, \boldsymbol{l})$ with polynomially bounded edge lengths, maintains a fully-dynamic $O(\log |V|)$-spanner H of G such that $|E(H)| \leq \gamma_{DS} \cdot |V(H)|$ and $\|H\|_{\to} \leq \gamma_{DS} \cdot (\|D_G\|_{\to} + |V|)$ for $\gamma_{DS} \in \widetilde{O}(1)$, where D_G is a dynamic object that contains all decremental updates to G (deletions and vertex splits).*

H can be maintained in polynomial time per update.

Proof. We assume unit lengths via maintaining separate spanners for length ranges $[2^i, 2^{i+1})$ and returning the direct product of the appropriately scaled spanners. Furthermore, we let $|V^{(0)}| = n$ and re-start after n decremental updates.

Then, there can be at most n vertex splits and therefore G contains at most $2n$ vertices before re-starting. We initialize $H^{(0)}$ to the greedy spanner of $G^{(0)}$, i.e. we consider the edges $(u, v) \in E(G^{(0)})$ in arbitrary order and add them to $E(H^{(0)})$ if $\text{dist}_{H^{(0)}}(u, v) > 2 \log 2n$. This ensure that the graph $H^{(0)}$ is a $O(\log |V|)$-spanner of $G^{(0)}$, and $|E^{(0)}| \leq O(n)$ directly follows from the fact that a girth $2 \log |V| + 1$ graph contains at most $O(|V|)$ edges.[1]

Whenever G is updated, we perform the corresponding operation on H in case of edge deletions and vertex splits such that $H \subseteq G$. Then, we iteratively check for every edge $(u, v) \in E(G) \setminus E(H)$ if $\text{dist}_H(u, v) > 2 \log 2n$ and insert the edges for which the condition is true. In particular, for the insertion of an edge (u, v), this corresponds to inserting the edge whenever $\text{dist}_H(u, v) > 2 \log 2n$.

Clearly H is an $O(\log n)$ spanner of G throughout because every edge is stretched by at most $2 \log 2n$. Furthermore, vertex splits and edge deletions only increase the girth of H, and therefore H contains at most $O(n)$ edges throughout. Finally, the total recourse bound $\|H\|_{\to} \leq O(n)$ follows until the data structure is re-started because an edge only leaves the graph H when it is deleted.

To conclude, it suffices to observe that the total number of restarts is bounded by $\|D_G\|_{\to} / n$ and there are $O(\log n)$ data structures in total, so $\gamma_{\text{DS}} = O(\log n)$. ◀

3.4 Low-Recourse Pivot Hierarchies via Iterated Sparsification

Given the low-recourse vertex and edge sparsification routines described above, we are ready to give the algorithm for Theorem 10.

Our algorithm consists of layers $0, \ldots, \Lambda - 1$ where $\Lambda = \sqrt{\log n}$.[2] Each layer consists of two graphs G_i and H_i, where $H_i \subseteq G_i$. We let $G_0 = G$. Then we obtain H_i from G_i and G_i from H_{i-1} as follows:

1. G_i is the output of the vertex sparsifier from Theorem 13 on the graph H_{i-1} for size reduction parameter $k \stackrel{\text{def}}{=} 2^{\sqrt{\log n}}$. For simplicity, we assume that k is an integer without loss of generality.
2. H_i is the dynamic spanner from Theorem 14 on the graph G_i.

[1] In [1] this folklore bound was already used to obtain a spanner.

[2] We remark that our full algorithm uses a much more drastic size reduction to enable bootstrapping, which results in only $O(\log^{\epsilon} \log n)$ layers for some small constant $\epsilon < 1$. This is necessary because the stretch is powered every time we recurse.

Whenever an update to G occurs, we pass the updates up the hierarchy until we reach the first layer j that has recieved more than $\gamma_{DS}^j \gamma_{VS}^j 2^{\Lambda-j}$ since it was initialized. Then, we re-initialize all the layers $j, \ldots, \Lambda$ via the steps described above. Notice in particular that the recourse of each layer is much smaller than the size reduction, leading to manageable recourse over all.

We let the sets A_i be the vertex sets of the graphs G_i mapped back to G, and we let A_Λ contain an arbitrary vertex r in $A_{\Lambda-1}$ (also mapped back to G). Furthermore, we let the pivot map $p_i(\cdot)$ be the pivot function maintained by the data structure from Theorem 13 at layer $i+1$ for $i = 0, \ldots, \Lambda-1$, and the cluster map C_i is the cluster map from vertex sparsifier data structure at layer $i+1$. Finally, we define the cluster map $C_\Lambda(v) \stackrel{\text{def}}{=} A_\Lambda$ and pivot map $p_\Lambda(v)$ for every vertex $v \in A_\Lambda$.

Proof of Theorem 10. We first bound the sizes and recourse of the sets $A_0, \ldots, A_\Lambda$ by induction. The recourse of the set A_0 and the recourse of G_0 is $\|G\|_\to$ by definition, and the recourse of H_0 is at most $\gamma_{\text{DS}}\big(\|G\|_\to + |V|\big)$.

We then observe that recourse caused by non-decremental updates on G_i can be ignored because they do not show up in the recourse of the subsequent spanner H_i by Theorem 14. The recourse of the decremental updates to G_i is $\gamma_{VS} \cdot \|H_{i-1}\|_\to$. The recourse of H_i is $\|H_i\|_\to \leq \gamma_{DS}(\|D_{G_i}\|_\to + |V(G_i)|)$. Therefore, t updates to G cause at most $O(t \cdot \gamma_{VS}^i \gamma_{DS}^i)$ vertices to be added to G_i. Since we re-start every layer i after $\gamma_{DS}^i \gamma_{VS}^i 2^{\Lambda-i}$ updates have arrived and the total amount of updates that reach layer i is $\gamma_{VS}^i \gamma_{DS}^i n$, the total recourse of the layer is $O(2^\Lambda \cdot n \cdot \gamma_{VS}^{i+1} \gamma_{DS}^{i+1})$ and the total recourse of all the sets is at most $O(k \cdot \gamma_{VS}^{\Lambda+1} \gamma_{DS}^{\Lambda+1} \cdot n \cdot \Lambda)$ by Theorem 13 and Theorem 14.

Then, we show that the final set $A_{\Lambda-1}$ is of small enough size for bounding the size of the clusters $C_{\Lambda-1}(\cdot)$. By the recourse bound above, there are at most $n^{o(1)} n / k^{\Lambda-i+1}$ updates to a level before it gets rebuilt. Therefore, the size of level $\Lambda-1$ is at most $n^{o(1)}$ at all times. The bound on the other cluster sizes directly follows from Theorem 13 and the description of our algorithm.

It remains to show that the distance estimates $\widetilde{\text{dist}}$ are good. To do so, we first notice that for every $u, v \in A_i$ we have $\text{dist}_G(u,v) \leq \text{dist}_{G_i}(u,v) \leq n^{o(1)} \text{dist}_G(u,v)$ since each layer only loses $\widetilde{O}(1)$ in distance approximation by Theorem 13 and Theorem 14 and there are $\Lambda = \sqrt{\log n}$ levels. Then, the distance approximation follows by induction since the length of the detour to the pivots can lose at most a poly-logarithmic factor per level because the pivot is closer in the graph G_i and $\text{dist}_G(u,v) \leq \text{dist}_{G_i}(u,v) \leq n^{o(1)} \cdot \text{dist}_G(u,v)$.[3] ◀

4 The Algorithm

We solve the APSP problem on a large graph G using a pivot hierarchy obtained by iterated vertex and edge sparsification, just like in the warm up in Section 3. Again, we will select a set of pivots for the first level of a pivot hierarchy, and perform edge and vertex sparsification to represent the distances between the pivots using a smaller dynamic graph, and then recurse on this graph. The main difference to the previous section is that (a) we want to bootstrap dynamic APSP along the way and (b) we want to develop and use a computationally efficient spanner. This spanner creates a few technical complications. In particular, to make the spanner efficient, we need all our graphs to have somewhat bounded maximum degree.

[3] The bounds on the detour are analogous to [17] after taking into account the extra distortion of the distances in G_i as compared to the original graph G. This extra distortion causes the sub-polynomial overhead.

4.1 Vertex sparsification

First, our algorithm reduces the APSP problem to a graph $\widetilde{G}$ with significantly fewer vertices using the KMG-vertex sparsifier (See Section 4.3). When compared to the low-recourse version presented in Section 3, our vertex sparsifier has an additional routine called ReduceDegree$(\cdot)$ which forces some of the vertices in the vertex sparsifier to be split and therefore have smaller degree. We further elaborate this point in the section below.

The vertex sparsification algorithm is presented in Section 4.3.

4.2 Edge sparsification

Although $\widetilde{G}$ is supported on a much smaller vertex set, it may still contain most of the edges from G. To achieve a proper size reduction, we additionally employ edge sparsification on top of the vertex sparsifier $\widetilde{G}$.

There are two key differences between the warm up and the efficient spanner algorithm.

- To quickly check for detours in the spanner, the efficient algorithm recursively uses APSP datastructures on small instances (See Section 4.4).
- To efficiently maintain the dynamic spanners, it is crucial that the degree of the graph remains bounded. Although we can assume that the degree of the initial graph G is bounded by a constant, repeatedly sparsifying the graph could seriously increase its maximum degree. Fortunately, our dynamic spanner guarantees that the average degree after edge sparsification is low. However, the same cannot be achieved for the maximum degree, which becomes apparent when trying to sparsify a star graph.
  To remedy this issue, we use that the underlying graph has bounded degree and call the ReduceDegree$(\cdot)$ routine of the KMG-vertex sparsifier to reduce the degree of high-degree vertices in the spanner. This operation can lead to changes in $\widetilde{G}$ and its spanner, but we show that repeatedly splitting vertices with high degree in the spanner quickly converges because the progress of splits is larger than the recourse they cause to the spanner.

The edge sparsification algorithm is presented in Section 4.4.

4.3 Pivots and Vertex Sparsification

We use the KMG-vertex sparsifier algorithm [16], which is based on dynamic core graphs, which in turn are based on static low-stretch spanning trees.

We first introduce certified pivots. These are mapping vertices that get sparsified away to vertices in $\widetilde{G}$, and certify the distance for vertices that are closer to each other than to their respective pivots.

▶ **Definition 15** (Certified Pivot). *Consider a graph $G = (V, E)$ with edge lengths $\boldsymbol{l}$ and a set $A \subseteq V$. We say that a function $p : V \to A$ is a pivot function for A, if for all $v \in V$, we have that $p(v)$ is the nearest vertex in A (ties broken arbitrarily but consistently).*

Additionally, we say that p is a certified pivot function if for all vertices $v \in V$, for all $u \in B_G(v, p(v)) \cup \{p(v)\}$ we have computed the exact (u, v) shortest paths and distances.

To motivate Definition 15, we show that pivots can be used to approximate the distance between sparsified vertices whenever their exact distance isn't stored already.

▷ Claim 16. Consider a graph $G = (V, E, l)$ with pivot set A and corresponding pivot function p. Then for any $u, v \in V$, if $v \notin B_G(u, p(u))$, we have $\mathrm{dist}_G(p(u), p(v)) \leq 4 \cdot \mathrm{dist}_G(u, v)$.

Proof. By $v \notin B_G(u, p(u))$ we directly obtain $\text{dist}_G(u, p(u)) \leq \text{dist}_G(u, v)$ and since $p(v)$ is (one of) the neareast vertices to v in A, we have $\text{dist}_G(v, p(v)) \leq \text{dist}_G(v, p(u)) \leq \text{dist}_G(v, u) + \text{dist}_G(u, p(u)) \leq 2\,\text{dist}_G(u, v)$ as well. By using the triangle inequality we get $\text{dist}_G(p(u), p(v)) \leq \text{dist}_G(p(u), u) + \text{dist}_G(u, v) + \text{dist}_G(v, p(v)) \leq 4\,\text{dist}_G(u, v)$. ◁

Next, we state a version of the KMG-vertex sparsifier [16], which is based on dynamic core graphs. We refer the reader to the full version of our article for a more detailed discussion of their vertex sparsifier.

The following theorem should be thought of as an efficient analog to Theorem 13 with the extra operation ReduceDegree(·). As mentioned at the start of this section, this routine comes into play when the spanner produces a high-degree vertex down the line.

▶ **Theorem 17** (KMG-vertex sparsifier, See Theorem 3.1 in [16])**.** *Consider a size reduction parameter $k > 1$ and an edge-dynamic graph G, with $\deg_{\max}(G) \leq \Delta$ and lengths in $[1, L]$. Then, there is a deterministic algorithm that explicitly maintains,*

1. *a monotonically growing pivot set $A \subseteq V(G)$ and certified pivot function $p : A \to V$, such that $|A| \leq n/k + 2\,\|G\|_{\to}$.*
2. *a fully-dynamic graph $\widetilde{G}$, such that $\widetilde{G}$ is a γ_{VS}-approximate vertex sparsifier of G with respect to A. We have that $\widetilde{G}^{(0)}$ has at most $m \cdot \gamma_{VS}$ edges and $\gamma_{VS} \cdot m/k$ vertices, where $\gamma_{VS} = \widetilde{O}(1)$. The following properties hold:*
   - **a.** *$\deg_{\max}(\widetilde{G}) \leq \Delta \cdot \gamma_{VS} \cdot k$, and $\widetilde{G}$ has lengths $\boldsymbol{l}_{\widetilde{G}}$ in $[1, nL]$, and*
   - **b.** *given any edge $e = (u, v) \in \widetilde{G}$, the algorithm can return a uv-path P in G with $l_G(P) \leq l_{\widetilde{G}}(e)$ in time $O(|P|)$.*
   - **c.** *a procedure ReduceDegree(E', z), where $E' \subseteq \{e \in E(\widetilde{G}) : v \in e\}$ for some $v \in V(\widetilde{G})$, and z is a positive integer. It updates the graph $\widetilde{G}$, performing at most $\gamma_{VS} \cdot |E'|/z$ vertex splits, such that afterwards any vertex $v' \in V(\widetilde{G})$ with $v \in \text{master}(v')$ is adjacent to at most $z \cdot \Delta$ edges in E'. This procedure runs in time at most $\gamma_{VS} \cdot k \cdot |E'|$.*
   - **d.** *If R_V is the dynamic object containing the total number of vertex splits performed by ReduceDegree, then $\left\|\widetilde{G}\right\|_{\to} \leq \gamma_{VS} \cdot \|G\|_{\to} + \|R_V\|_{\to}$.*

The algorithm runs in time $m \cdot \gamma_{VS} \Delta k^4 + \|G\|_{\to} \cdot \gamma_{VS} k^4 \Delta$.

4.4 Edge Sparsification

We state our main dynamic spanner theorem whose proof is deferred to the full version of our article. The theorem assumes access to a $\gamma_{apxAPSP}$-approximate, $\widetilde{O}(1)$-query APSP data structure (See Definition 7.)

▶ **Theorem 18** (Dynamic Spanner)**.** *There exists a data structure that given a fully-dynamic graph G with unit lengths and $|V(G^{(0)})| = n$, $\|G\|_{\to} \leq n$ and a parameter $1 \leq K \leq O(\log^{1/3} n)$, maintains a fully-dynamic $\gamma_l^{O(K)}$-spanner S of G such that $|E(S)| \leq \gamma_{ES} \cdot n$ and $\|S\|_{\to} \leq \gamma_{ES} \cdot \|G\|_{\to}$, where $\gamma_l = \widetilde{O}(\gamma_{apxAPSP})$ and $\gamma_{ES} = \widetilde{O}(n^{1/K})$. It runs in time $n \deg_{\max}(G^{(0)}) \gamma_{ES} \gamma_l^{O(K^2)} + APSP(\gamma_{ES} n, 3, \gamma_{ES} n)$.*

▶ Remark 19. We can directly extend Theorem 18 to graphs with edge lengths in $[1, L]$ by bucketing edges in the intervals $[2^i, 2^{i+1})$ for $i = 1, \ldots, O(\log L)$.

Note also that as at all times H is a subgraph of G, it does not undergo more vertex splits and isolated vertex insertions than G, i.e., all extra updates to maintain H are edge insertions/deletions.

4.5 Bootstrapping via Edge and Vertex Sparsification

We are given an edge-dynamic graph G on initially n vertices and m edges, that at all times has maximum degree at most Δ. Applying Theorem 17 to G reduces the problem to finding short paths in $\widetilde{G}$, a graph of $\widetilde{O}(m/k)$ vertices. Although the number of vertices significantly decreases, the graph still contains up to $\widetilde{O}(m)$ edges, preventing efficient recursion. To address this, we apply Theorem 18 on $\widetilde{G}$ to produce a graph H on $\gamma \cdot m/k$ vertices *and* edges, where γ crucially depends only on the parameter K of Theorem 18.

Note that the APSP data structure that Theorem 18 requires also only needs to run on a graph of at most $\gamma \cdot m/k$ vertices and edges, so by choosing a large enough size reduction factor k, the problem size reduces by a factor of γ/k, while the approximation factor increases from $\gamma_{apxAPSP<m}$ to $\widetilde{O}(\gamma_{apxAPSP<m})^{O(K)}$.[4] Balancing both parameters k and K allows us to bootstrap an APSP data structure with the guarantees from Theorem 1.

While H is sparse, it might still have high maximum degree, preventing efficient recursion. The sparsity still implies, however, that there cannot be too many high degree vertices. Hence, we carefully perform a few vertex splits in $\widetilde{G}$ by leveraging the ReduceDegree$(\cdot)$ functionality from Theorem 17 to enforce that H also has small maximum degree.

Algorithm 1 initDynamicAPSP().

1 Maintain $\widetilde{G}$ from Theorem 17.
2 Let H be the graph maintained by applying Theorem 18 to graph $\widetilde{G}$ with parameter K.
`/* Here` $\gamma_{degConstr}$ `is a constant fixed later. */`
3 **while** $\exists v \in V(H)$ *with* $\deg_H(v) > 8\gamma_{degConstr} \cdot \Delta$ **do**
4 | ReduceDegree$(E_H(v), \gamma_{degConstr})$.
5 Initialize and maintain a recursive dynamic APSP data structure on H.

The recursive APSP instance that is running on H expects an edge-dynamic graph, while the spanner from Theorem 18 is fully-dynamic. To circumvent this issue, we simulate the vertex splits by suitable edge insertions/removals and vertex insertions. As the maximum degree of H gets controlled during the while loops, this simulation can lead to at most an extra factor of $O(\gamma_{degConstr} \cdot \Delta)$ in the number of updates.

Algorithm 2 MaintainDynamicAPSP(G, t).

1 Update $\widetilde{G}$ and H accordingly.
2 **while** $\exists v \in V(H)$ *with* $\deg_H(v) > 8\gamma_{degConstr} \cdot \Delta$ **do**
3 | ReduceDegree$(E_H(v), \gamma_{degConstr})$.
4 | Update H accordingly.
5 Forward all updates made to H to the recursive dynamic APSP datastructure that runs on H by simulating the vertex splits accordingly

When queried for the distance between two vertices u and v, our algorithm checks if the certified pivot function stores the distance, and if this is not the case it recursively asks the smaller instance DynamicAPSP$_H$ for the distance between $p(u)$ and $p(v)$ in H.

[4] We only have a valid size reduction if also $\gamma \cdot m/k < n$. We will see that this is indeed the case as we can assume that initially $m = O(n)$.

Algorithm 3 $\textsc{Dist}(u, v)$.

```
if v ∈ B_G(u, A) then
    return dist(u, v)
else
    return dist(u, p(u)) + DynamicAPSP_H.Dist(p(u), p(v)) + dist(p(v), v)
```

4.6 Analysis

We first show that the while loops in Algorithm 1 and Algorithm 2 terminate sufficiently fast. This is the most crucial claim for the analysis.

▷ Claim 20. If R_V is the dynamic object counting the total number of vertex splits performed by ReduceDegree() in Algorithm 1 and Algorithm 2, then $\|R_V\|_{\to} \leq 2\gamma_{VS}(m/k + \|G\|_{\to})$. In particular, the while loops always terminate and immediately thereafter H has maximum degree at most $8\gamma_{degConstr} \cdot \Delta$. The total runtime of all calls to ReduceDegree() is $6\gamma_{VS}^2\gamma_{ES}(m + k\|G\|_{\to})$.

Proof. For tracking progress achieved by vertex splits, we introduce the following potential function

$$\Phi(H) \stackrel{\text{def}}{=} \sum_{v \in V(H)} \max\{\deg_H(v) - \Delta \cdot \gamma_{degConstr}, 0\}.$$

Let t be the time before we enter the while loop in Algorithm 1. Then by Theorem 17 we have that $|V(\widetilde{G}^{(t)})| \leq \gamma_{VS}m/k$ and thus by Theorem 18, we have that $|E(H^{(t)})| \leq \gamma_{VS}\gamma_{ES}m/k$.

By the Handshake-Lemma, $\Phi(H^{(t)}) \leq 2|E(H^{(t)})| \leq 2 \cdot \gamma_{ES} \cdot \gamma_{VS} \cdot m/k$. Note that whenever an update is made to H, the potential can increase by at most 2. This is as edge deletions and vertex splits can only decrease it, and an edge insertion can increase it by at most 2.

Let R_E be the dynamic object counting the number of edges that were passed to ReduceDegree, and note that if $\textsc{ReduceDegree}(E', \gamma_{degConstr})$ is called, we have $|E'| > 8\gamma_{degConstr} \cdot \Delta$ by the while loop condition.

By the properties of ReduceDegree() in Theorem 17, we have that $\|R_V\|_{\to} \leq \gamma_{VS}/\gamma_{degConstr} \cdot \|R_E\|_{\to}$, and that none of the vertices v' that result from the splits to v will be adjacent to more than $\gamma_{degConstr} \cdot \Delta$ of the edges in E'. So if we just forwarded these splits to H, we would have $\deg_H(v') \leq \gamma_{degConstr} \cdot \Delta$, and the potential decreases by at least $7/8 \cdot |E'|$.

However, forwarding these vertex splits to H, Theorem 18 will cause further updates to H to maintain the spanner. Fortunately by Item 2d of Theorem 17 we know that $\|H\|_{\to} \leq \gamma_{ES} \cdot \left\|\widetilde{G}\right\|_{\to} \leq \gamma_{ES} \cdot \gamma_{VS} \cdot \|G\|_{\to} + \gamma_{ES} \cdot \|R_V\|_{\to}$. Let $\gamma = \gamma_{ES} \cdot \gamma_{VS}$. Putting everything together we show that the potential decreases as follows.

$$\begin{aligned}\Phi(H) &\leq 2\gamma m/k - \frac{7}{8}\|R_E\|_{\to} + 2\|H\|_{\to} \leq 2\gamma m/k - \frac{7}{8}\|R_E\|_{\to} + 2\gamma\|G\|_{\to} + 2\gamma_{ES}\|R_V\|_{\to} \\ &\leq 2\gamma m/k - \frac{7}{8}\|R_E\|_{\to} + 2\gamma\|G\|_{\to} + \frac{2\gamma_{ES}\gamma_{VS}}{\gamma_{degConstr}}\|R_E\|_{\to}.\end{aligned}$$

Thus, choosing $\gamma_{degConstr} = \lceil 4\gamma_{VS}\gamma_{ES} \rceil$, we get $\Phi(H) \leq 2\gamma m/k - \frac{3}{8}\|R_E\|_{\to} + 2\gamma\|G\|_{\to}$. As $0 \leq \Phi(H)$, this directly implies that $\|R_E\|_{\to} \leq 6\gamma_{ES}\gamma_{VS}(m/k + \|G\|_{\to})$. Remembering that $\|R_V\|_{\to} \leq \frac{\gamma_{VS}}{\gamma_{degConstr}}\|R_E\|_{\to}$ then yields the bound on $\|R_V\|_{\to}$.

Since the total runtime is given by $\gamma_{VS} \cdot k \cdot \|R_E\|_{\to}$, this bound follows directly. ◁

We then show a bound on the total runtime of our dynamic APSP algorithm in terms of costs for smaller instances, setting us up for our final recursion.

To do so, assume the recursive APSP instance that is running on H and used in Theorem 18 is a $\gamma_{apxAPSP<m}$-approximate, $\widetilde{O}(1)$-query APSP. Let $\gamma_l = \widetilde{O}(\gamma_{apxAPSP<m})$ be the stretch parameter from Theorem 18.

▶ **Theorem 21.** *The total combined runtime of Algorithm 1 and Algorithm 2 over a sequence of up to m/k updates is $m \cdot \gamma_{RC} \cdot \gamma_l^{O(K^2)} \Delta k^4 + APSP(\gamma_{RC} m/k, \gamma_{RC}\Delta, \gamma_{RC}\Delta m/k)$ for some $\gamma_{RC} = \widetilde{O}(m^{3/K})$.*

Proof. We first show that for at most m/k updates, initializing and maintaining H is efficient. Note that by the previous claim, we can bound the total costs incurred in the while loops by $6\gamma_{VS}^2\gamma_{ES}(m + k\|G\|_{\to}) \leq 12\gamma_{VS}^2\gamma_{ES}m$. The total costs of initializing and maintaining $\widetilde{G}$ are $2m \cdot \gamma_{VS}\Delta k^4$ by Theorem 17. Finally, by the previous lemma $\|\widetilde{G}\| \leq \gamma_{VS}\|G\|_{\to} + \|R_V\|_{\to} \leq 3\gamma_{VS}\|G\|_{\to} + 2\gamma_{VS}m/k \leq 4\gamma_{VS}m/k$, so that we can further bound the runtime of Theorem 18 by $\gamma_{VS}^2\gamma_{ES}n\Delta k\gamma_l^{O(K^2)} + APSP(\gamma_{ES} \cdot n, 3, \gamma_{ES} \cdot n)$. Note that here we also crucially used that by Theorem 17, $\Delta_{\max}(\widetilde{G}) \leq k\gamma_{VS}\Delta$.

It remains to consider the APSP data structure that we have to run on H. First note that by the gurantees of Theorem 17 and Theorem 18, initially $|E(H)| \leq \gamma_{VS}\gamma_{ES}m/k$. Secondly, note that as $\|\widetilde{G}\| \leq 4\gamma_{VS}m/k$, we in particular have $\|H\|_{\to} \leq 4\gamma_{ES}\gamma_{VS}m/k$. However, as H is vertex dynamic, we also have to simulate all the vertex splits occuring in H, which can lead to up to $8\gamma_{degConstr}\Delta$ extra updates per split. Hence the total amount of updates that the APSP data structure receives are up to $O(\gamma_{VS}^2\gamma_{ES}^2 m/k)$. By the while loop conditions, we do know that the maximum degree of the graph is always bounded by $8\gamma_{degConstr}\Delta$. Therefore, the total runtime bound follows. ◀

We show that the stretch does not increase too much as the size of the graph increases.

▷ Claim 22. Algorithm 1 provides a worst case stretch of $\widetilde{O}(\gamma_{apxAPSP<m})^{O(K)}$.

Proof. If $v \in B_G(u, A)$ there is nothing to show as then by Theorem 17 we have already pre-computed the exact distance between u and v.

But if $v \notin B_G(u, A)$, then by Claim 16, we have that $\text{dist}_G(p(u), p(v)) \leq 4 \cdot \text{dist}_G(u, v)$. As $p(u), p(v) \in A$, we have $\text{dist}_H(p(u), p(v)) \leq \gamma_l^{O(K)} \cdot \text{dist}_{\widetilde{G}}(p(u), p(v)) \leq \gamma_l^{O(K)} \cdot \gamma_{VS} \cdot \text{dist}_G(p(u), p(v)) = \widetilde{O}(1) \cdot (\widetilde{O}(1) \cdot \gamma_{apxAPSP})^{O(K)} \cdot \text{dist}_G(p(u), p(v)) = \widetilde{O}(\gamma_{apxAPSP})^{O(K)} \cdot \text{dist}_G(p(u), p(v))$. ◁

▶ Remark 23. We can directly extend this to receive a procedure $\textsc{Path}(u, v)$ that provides a u-v path P with $|P| \leq \widetilde{O}(\gamma_{apxAPSP<m})^{O(K)} \cdot \text{dist}_G(u, v)$ and runs in time $\widetilde{O}(|P|)$. This is because Theorem 17 also explicitly stores exact shortest paths between vertices u and their pivots $p(u)$.

In order to prove Theorem 1, we need to carefully assemble the previous insights, which we defer to the full version of our article.

References

1 Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9(1):81–100, 1993. doi:10.1007/BF02189308.

2 A. Bernstein, M. Gutenberg, and T. Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1000–1008, Los Alamitos, CA, USA, February 2022. IEEE Computer Society. doi:10.1109/FOCS52979.2021.00100.

3 Sayan Bhattacharya, Thatchaphol Saranurak, and Pattara Sukprasert. Simple Dynamic Spanners with Near-Optimal Recourse Against an Adaptive Adversary. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms (ESA 2022)*, volume 244 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2022.17.

4 J. Brand, L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. Gutenberg, S. Sachdeva, and A. Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 503–514, Los Alamitos, CA, USA, November 2023. IEEE Computer Society. doi:10.1109/FOCS57990.2023.00037.

5 Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 170–181, 2018. doi:10.1109/FOCS.2018.00025.

6 Li Chen, Rasmus Kyng, Yang P. Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, s-t shortest path, and minimum-cost flow. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, pages 1165–1173, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649745.

7 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623, 2022. doi:10.1109/FOCS54457.2022.00064.

8 Julia Chuzhoy. Decremental all-pairs shortest paths in deterministic near-linear time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, pages 626–639, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3406325.3451025.

9 Julia Chuzhoy and Thatchaphol Saranurak. Deterministic Algorithms for Decremental Shortest Paths via Layered Core Decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Proceedings, pages 2478–2496. Society for Industrial and Applied Mathematics, January 2021. doi:10.1137/1.9781611976465.147.

10 Julia Chuzhoy and Ruimin Zhang. A new deterministic algorithm for fully dynamic all-pairs shortest paths. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, pages 1159–1172, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3564246.3585196.

11 Sebastian Forster, Gramoz Goranci, and Monika Henzinger. Dynamic maintenance of low-stretch probabilistic tree embeddings with applications. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1226–1245, 2021. doi:10.1137/1.9781611976465.75.

12 Sebastian Forster, Gramoz Goranci, Yasamin Nazari, and Antonis Skarlatos. Bootstrapping Dynamic Distance Oracles. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 50:1–50:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.50.

13 Sebastian Forster, Yasamin Nazari, and Maximilian Probst Gutenberg. Deterministic incremental apsp with polylogarithmic update time and stretch. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, pages 1173–1186, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3564246.3585213.

14 Bernhard Haeupler, Yaowei Long, and Thatchaphol Saranurak. Dynamic deterministic constant-approximate distance oracles with n^{ϵ} worst-case update time, 2024. doi:10.48550/arXiv.2402.18541.

15 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. *J. ACM*, 65(6), November 2018. doi:10.1145/3218657.

16 Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. A dynamic shortest paths toolbox: Low-congestion vertex sparsifiers and their applications. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, pages 1174–1183, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649767.

17 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, January 2005. doi:10.1145/1044731.1044732.