A Simple Dynamic Spanner via APSP

Rasmus Kyng
ETH Zurich, Switzerland

Simon Meierhans
ETH Zurich, Switzerland

Gernot Zöcklein
ETH Zurich, Switzerland

Abstract

We give a simple algorithm for maintaining a $n^{o(1)}$-approximate spanner H of a graph G with n vertices as G receives edge updates by reduction to the dynamic All-Pairs Shortest Paths (APSP) problem. Given an initially empty graph G, our algorithm processes m insertions and n deletions in total time $m^{1+o(1)}$ and maintains an initially empty spanner H with total recourse $n^{1+o(1)}$. When the number of insertions is much larger than the number of deletions, this notably yields recourse sub-linear in the total number of updates.

Our simple algorithm can be extended to maintain a $\delta \geq \omega(1)$-approximate spanner with $n^{1+o(1)}$ edges throughout a sequence of m insertions and D deletions with amortized update time $n^{o(1)}$ and total recourse $n^{1+o(1)} + n^{o(1)} \cdot D$ via batching.

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

Keywords and phrases Dynamic graph algorithms, Spanner, Dynamic Greedy Spanner

Digital Object Identifier 10.4230/LIPIcs.ICALP.2025.111

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2408.11368`

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.

Acknowledgements The authors are very grateful for feedback from Maximilian Probst Gutenberg and Pratyai Mazumder on a draft of this article that improved the presentation.

1 Introduction

Given a graph $G = (V, E)$, a δ-approximate spanner H is a subgraph of G with the same vertex set such that $\text{dist}_H(u, v) \leq \delta \cdot \text{dist}_G(u, v)$ for every pair of vertices $u, v \in V$.[1] Typically, the spanner H has much fewer edges than G and can therefore be used as a sparse proxy for computing approximate distances in downstream applications. Althöfer, Das, Dobkin, Joseph and Soares developed the now-classical greedy spanner [1].

Given an integer δ, their algorithm incrementally constructs a $(2\delta - 1)$-approximate spanner H of $G = (V, E)$ as follows.
1) Initialize $H = (V, \emptyset)$.

[1] δ is called the stretch of H.

52nd International Colloquium on Automata, Languages, and Programming (ICALP 2025).
Editors: Keren Censor-Hillel, Fabrizio Grandoni, Joël Ouaknine, and Gabriele Puppis
Article No. 111; pp. 111:1–111:11
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2) Consider the edges $(u, v) \in E$ in arbitrary order. If $\text{dist}_H(u, v) \geq 2\delta$, add (u, v) to H. When this process finishes, every edge on the shortest path between two arbitrary vertices in G can be replaced by a detour of length at most $2\delta - 1$ in H. The claimed approximation $\text{dist}_H(u, v) \leq (2\delta - 1) \cdot \text{dist}_G(u, v)$ directly follows.

Crucially, they show that the graph H is also sparse. This follows from the folklore observation that high-girth graphs are sparse. The girth of a graph is the length of the shortest cycle. By construction, the girth of H is $2\delta + 1$, which implies total edge count is bounded by $2|V|^{1+\frac{1}{\delta}}$. Whenever δ grows with n, this leads to an almost-linearly sized spanner.

▶ **Observation 1** (Folklore). *Every graph $H = (V, E_H)$ with girth $2\delta + 1$ contains at most $2|V|^{1+\frac{1}{\delta}}$ edges.*

Proof. Repeatedly remove vertices of degree at most $d = |V|^{\frac{1}{\delta}} + 1$ from H. This process removes at most $d \cdot |V|$ edges. Assume for the sake of a contradiction that there is some remaining graph $\widetilde{H} \subseteq H$ after this process terminates. Fix an arbitrary remaining vertex r, and consider the breath-first search tree $T \subseteq \widetilde{H}$ to depth δ from r. Since the girth of $\widetilde{H}$ is at least $2\delta + 1$, no two vertices at depth $< \delta$ in T share an edge in $\widetilde{H} \setminus T$. Thus, the number of vertices in T is at least $\sum_{i=0}^{\delta}(d-1)^i > (d-1)^\delta = |V|$ which is a contradiction. ◀

Despite the simplicity of this framework, previous deterministic *dynamic algorithms* for maintaining a spanner $H \subseteq G$ as G undergoes updates were based on expander decompositions [7, 5].

However, In [4] the authors observed that the greedy spanner construction can be used to maintain a spanner $H \subseteq G$ as G receives edge deletions with low amortized recourse. As part of their almost-linear time incremental min-cost flow algorithm, [6] presented a computationally efficient algorithm for maintaining a spanner of a fully-dynamic graph G based on the greedy spanner.[2] Their algorithm is complicated by the fact that they need to process vertex splits in addition to edge updates and the total number of decremental updates is limited by the number of vertices.

1.1 A simple dynamic spanner for graphs with few deletions

We present a simplified algorithm for maintaining a spanner of an edge-dynamic graph by reduction to an All-Pairs Shortest Paths (APSP) oracle. The underlying APSP oracle is a dynamic algorithm providing query access to approximate pairwise distances in a dynamic graph [9, 12, 11, 13].

▶ **Theorem 2.** *Given an initially empty edge-dynamic graph $G = (V, E)$ on $n = |V|$ vertices receiving m edge insertions and up to n edge deletions where $m \geq n$, there is an algorithm maintaining a $n^{o(1)}$-approximate spanner H with total update time $m^{1+o(1)}$ and total recourse $n^{1+o(1)}$.*

We note in particular that the total recourse is *almost-optimal* and only depends on the number of vertices and deletions, and is therefore completely independent of the number of insertions.

[2] They crucially exploit that the recourse is sub-linear in the number of insertions. Expander decomposition based algorithms seem less suitable for achieving such a result.

Our simple algorithm yielding Theorem 2 is presented in Section 4. It can use any path-reporting approximate APSP data structure and only adds an $O(\log n)$ factor overhead in runtime and approximation. We refer the reader to Definition 5 and Theorem 12 for a more fine grained analysis of the dependency on the underlying APSP data structure.

▶ Remark 3. Theorem 2 can be extended to graphs with polynomially bounded edge lengths via standard length bucketing. The number of edges in the final spanner increases by a logarithmic factor.

1.2 More deletions and lower stretch dynamic spanners

Batching techniques allow us to extend our spanner to handle an arbitrary number of deletions, where the recourse becomes almost-linear in the number of deletions and vertices.

An originally independent work [8] recently introduced the idea of obtaining a dynamic spanner with stretch $\delta = o(\log n)$.[3] They remark that prior to their work, no algorithm with update time $o(n)$, $n^{1+o(1)}$ edges and stretch $o(\log n)$ existed. This motivated us to extend our spanner to handle arbitrary non-constant stretch as in Theorem 18. They heavily use ideas from [11] directly, which we completely black-box. We remark that our algorithm loses large sub-polynomial factors in update time and density compared to theirs.

▶ **Theorem 4.** *Given an edge-dynamic graph $G = (V, E)$ with $n = |V|$ vertices receiving m insertions and D deletions, and a parameter $\delta \geq \omega(1)$, there is an algorithm maintaining a δ-approximate spanner H containing at most $n^{1+o(1)}$ edges with total recourse $n^{1+o(1)} + n^{o(1)} \cdot D$ and amortized update time $n^{o(1)}$.*

Our algorithm yielding Theorem 4 is presented in Section 5. It depends on the APSP data structure of [11], which can get arbitrarily good approximation by trading it off for update time. We point out that the recourse achieved by our algorithm is again *almost-optimal*, since scaling linearly in the number of deletions is unavoidable.

1.3 Prior Work

In [2], the authors present a randomized dynamic spanner with poly-logarithmic stretch and amortized update time against an oblivious adversary.[4] Then [10] significantly reduced the poly-logarithmic overhead in both size and update time. In [3], the authors gave the first non-trivial algorithm for maintaining a dynamic spanner against an adaptive adversary. They maintain a spanner of near linear size with poly-logarithmic amortized update time and stretch. Deterministic algorithms for maintaining spanners of dynamic graphs with bounded degree that additionally undergo vertex splits were developed in the context of algorithms for minimum cost flow [7, 5, 6]. These have sub-polynomial overhead.

[3] The first version of this article on arXiv precedes the announcement of [8], but their article prompted us to add Theorem 4 and Section 5.

[4] Oblivious adversaries generate the sequence of updates at the start, i.e. independently from the random output of the algorithm. Adaptive adversaries can use the previous output to fool a randomized algorithm.

2 Preliminaries

2.1 Static and Dynamic Graphs

We let $G = (V, E)$ denote unweighted and undirected graphs with vertex set V and edge set E. When we want to refer to the vertex/edge set of some graph G, we sometimes use $V(G)$ and $E(G)$ respectively.

We refer to $|E(G)| / |V(G)|$ as the density of a graph, and say a graph is sparse whenever its density is $n^{o(1)}$.

For a graph $G = (V, E)$ and a pair of vertices $u, v \in V$ we let $\text{dist}_G(u, v)$ denote the length of a shortest uv-path in G. We call a graph edge-dynamic if it receives arbitrary updates to E (i.e. edge insertions and deletions). We refer to the total number of such edge updates a graph receives as the recourse.

2.2 Edge Embeddings

Given two graph H and G on the same vertex set such that $H \subseteq G$, we let $\Pi_{G \to H}$ be a function that maps edges in $E(G)$ to paths in H. Given an embedding $\Pi_{G \to H}$, we define the edge congestion $\text{econg}(\Pi_{G \to H}, e) \stackrel{\text{def}}{=} |\{f \in E(G) : e \in \Pi_{G \to H}(f)\}|$ for $e \in E(H)$.

3 Dynamic All-Pairs Shortest Paths

We first define approximate *All-Pairs Shortest Paths* (APSP) data structures for edge-dynamic graphs.

▶ **Definition 5** (APSP). *For an initially empty edge-dynamic graph $G = (V, E)$ a γ-approximate α-update β-query APSP data structure supports the following operations.*

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

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

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

We refer the reader to [9, 12, 11, 13] for recent results on path-reporting dynamic APSP. The parameters γ, α and β should currently be thought of as sub-polynomial factors $n^{o(1)}$ in the number of vertices n of type $e^{O(\log^\tau n)}$ for some constant $0 < \tau < 1$.

In [13] the authors present the shortest APSP algorithm yet by simplifying the algorithm of [12] at the expense of a larger sub-polynomial overhead in runtime, i.e. $\alpha = e^{1/(\log \log m)^c}$ for some $c > 0$. Their algorithm internally bootstraps a more complicated version of the spanner presented in this article that is tailored to their application.

The algorithm of [11] can obtain a better approximation $\gamma = \log^\epsilon n$ at the cost of large sub-polynomial update times $\alpha = n^{1/(\log \log n)^{\Theta(\epsilon)}}$.

4 A Dynamic Greedy Spanner

4.1 A Simple Low-Recourse Algorithm

We first present a very simple dynamic variant of the greedy spanner that merely limits the number of changes to H, i.e. its recourse. Then we describe the necessary changes for obtaining an efficient version using a dynamic APSP datastructure.

Let $G = (V, E)$ be an initially empty graph on n vertices. We maintain a $O(\log n)$-spanner $H \subseteq G$ as follows.

- When an edge (u, v) is inserted to G, we check if $\text{dist}(u, v) > 2 \log n$. If so, we add it to H.
- When an edge (u, v) is deleted from G, we remove it from H if it is present and re-insert all edges in $E(G) \setminus E(H)$ using the procedure described above. If (u, v) is not in H, we do not need to update our spanner.

This algorithm still maintains that the girth of H is at least $2 \log n + 1$, and therefore the graph H contains at most $O(n)$ edges at any point. Edges only leave the spanner H when they are deleted. If at most D edges get deleted throughout, a total recourse bound of $O(n + D)$ follows directly since the total recourse is bounded by (maximum # edges in H) + (# edges leaving H).

4.2 A Simple Efficient Algorithm

To turn the above framework into an efficient algorithm, we maintain an embedding $\Pi_{G \to H}$ that maps each edge in G to a short path in H and thus certifies that such a path still exists. We then only have to re-insert all edges whose embedding paths use the deleted edge. To bound the number of re-insertions, we carefully manage the edge congestion of $\Pi_{G \to H}$. Finally, we use an APSP data structure (See Definition 5) to obtain distances and paths. In the following, we assume there are at most m insertions, n deletions and that $|V| = n$.

Our algorithm internally maintains two graphs: The current spanner H and a not yet congested sub-graph $\widehat{H} \subseteq H$. We maintain that the edge congestion is strictly less than K for every edge in $\widehat{H}$, and maintain access to distances and paths in $\widehat{H}$ via a γ-approximate α-update β-query APSP data structure $\mathcal{D}_{\widehat{H}}$.

- When an edge e is inserted, we check if $\mathcal{D}_{\widehat{H}}$ returns distance at most $2\gamma \log n$ between the endpoints. If this is the case, we query it for a witness path P and store P as the embedding path of e. Then we check if any of the edges on the path now have K embedding paths using them, and if so we remove them from $\widehat{H}$. Otherwise, we add the edge e to H and $\widehat{H}$.
- When an edge e is deleted, we remove it from $\widehat{H}$ and H and re-insert all edges that now have a broken embedding path using the procedure above.

Choosing the threshold $K = m/n$ allows us to tolerate up to n deletions which yield m additional calls to the insertion procedure. See Algorithm 1 for detailed pseudocode.

4.3 Analysis

We first show that H is a $(2 \cdot \gamma \cdot \log n)$-approximate spanner.

▶ **Lemma 6.** *The graph H maintained by* DynamicSpanner() *(Algorithm 1) is a $(2 \cdot \gamma \cdot \log n)$-approximate spanner throughout.*

Algorithm 1 DynamicSpanner().

```
1  procedure Initialize(n, m, K)
2      G ← (V, ∅); H ← (V, ∅); Ĥ ← (V, ∅); Π_{G→H} ← ∅ ;        // Let V = {1, ..., n}.
3      Let D_Ĥ be a γ-approximate α-update β-query APSP datastructure (See
         Definition 5).
4  procedure InsertEdge(e = (u, v))
5      G ← G ∪ {e}
6      if D_Ĥ.ApxDist(u, v) ≤ γ · 2 · log n then
7          P ← D_Ĥ.Path(e); Π_{G→H}(e) ← P;
8          foreach f ∈ P do
9              if econg(Π_{G→H}, f) ≥ K then
10                 Ĥ ← Ĥ \ f; D_Ĥ.RemoveEdge(f);
11     else
12         H ← H ∪ (e); Ĥ ← H ∪ (e); Π_{G→H}(e) ← (e); D_Ĥ.AddEdge(e);
13 procedure DeleteEdge(e = (u, v))
14     G ← G \ e;
15     if e ∈ H then
16         D_Ĥ.RemoveEdge(e); H ← H \ e; Ĥ ← Ĥ \ e;
17         F ← {f ∈ E(G) : e ∈ Π_{G→H}(f)} ;      // Edges with broken embedding
             paths.
18         G ← G \ F ;                  // Temporarily remove edges in F from G
19         foreach f ∈ F do
20             Π_{G→H}(f) ← ∅ ;                        // Delete broken embeddings.
21         foreach f ∈ F do
22             InsertEdge(f) ;                            // Re-insert edges in F.
```

Proof. When an edge $e \in E(G)$ is inserted, it afterwards has a valid embedding path of length at most $2 \cdot \gamma \cdot \log n$ by the description of InsertEdge(e). Therefore the edge has a valid embedding path at this point. However, if any edge on this path is deleted, the edge e is re-inserted by the description of DeleteEdge(e). This establishes the claim. ◀

We then prove that the total number of times the routine InsertEdge(e) is called is bounded by $2m$.

▷ Claim 7. For all $f \in E(H)$, $\text{econg}(\Pi_{G \to H}, f) \leq K$ throughout.

Proof. Whenever the edge congestion of an edge in $\widehat{H}$ reaches K it is removed from $\widehat{H}$ and $\mathcal{D}_{\widehat{H}}$. By the description of InsertEdge() such an edge is never used in embedding paths again. Therefore the edge congestion of $\text{econg}(\Pi_{G \to H}, f)$ is bounded by K for every edge. ◁

▶ **Corollary 8.** *A sequence of at most m external insertions and m/K deletions causes at most $2m$ calls to* InsertEdge() *in total.*

Proof. By Claim 7, the edge congestion is at most K throughout. Thus, there are at most m additional calls to InsertEdge() issued by the DeleteEdge() routine since the algorithm re-inserts every edge with a broken embedding path. The corollary follows. ◀

Given the previous corollary, we are ready bound the total recourse of H.

▷ Claim 9. $|E(H)| \leq O(\frac{\gamma \cdot m \log n}{K} + n)$ throughout a sequence of m insertions and m/K deletions.

Proof. We bound the edges in $E(H) \setminus E(\widehat{H})$ and in $E(\widehat{H})$ separately, starting with the former. These edges were congested by K paths when they were removed from $\widehat{H}$. Since the total number of edges ever added is $2m$, the total cumulative congestion that all the edges ever experience is at most $4 \cdot m \cdot \gamma \cdot \log n$ by the description of InsertEdge(). Therefore, the total number of edges in $E(H) \setminus E(\widehat{H})$ is at most $(4 \cdot m \cdot \gamma \cdot \log n)/K$. Finally, by Definition 5 the graph $\widehat{H}$ has girth at least $2 \log n + 1$, and therefore at most $O(n)$ edges by Observation 1. ◁

▶ **Corollary 10.** *The recourse of H is $O(\frac{\gamma \cdot m \log n}{K} + n)$.*

Proof. Follows from Claim 9 since at most m/K edges get removed from H. ◀

It remains to bound the total time spent processing updates.

▶ **Lemma 11.** *The algorithm processes m insertions and m/K deletions in total time $O(\beta \cdot \gamma \cdot m \cdot \log n + \alpha \cdot \gamma \cdot \frac{m}{K} \cdot \log n + \alpha \cdot n)$.*

Proof. There are at most $2m$ calls to InsertEdge(). Each such call causes a distance and possibly a path reporting query to $\mathcal{D}_{\widehat{H}}$. Whenever a path reporting query is called, the resulting path P has length at most $2\gamma \cdot \log n$. This causes a total runtime of $O(\beta \cdot \gamma \cdot m \cdot \log n)$.

Then, by Corollary 10, at most $O(\frac{\gamma \cdot m \log n}{K} + n)$ edges get added and therefore also removed from $\widehat{H}$. These cause $O(\frac{\gamma \cdot m \log n}{K} + n)$ update calls to $\mathcal{D}_{\widehat{H}}$. This causes total runtime $O(\alpha \cdot \gamma \cdot \frac{m}{K} \cdot \log n + \alpha \cdot n)$.

All other operations can be subsumed in the times above. ◀

Our first theorem then follows directly after setting $K = m/n$.

▶ **Theorem 12** (Simple Spanner to APSP Reduction)**.** *Given a γ-approximate α-update β-query APSP data structure (Definition 5), there is an algorithm that maintains a $2\gamma \log n$-approximate spanner H of an initially empty graph G on n vertices throughout a sequence up to m insertions and n deletions in total time $O(\beta \cdot \gamma \cdot m \cdot \log n + \alpha \cdot \gamma \cdot n \cdot \log n)$. The total recourse of H is $O(\alpha \cdot \gamma \cdot n \log n)$.*

Proof. Follows from Corollary 10 and Lemma 11 with $K = m/n$. ◀

Then, Theorem 2 follows from Theorem 12 in conjunction with any of the APSP data structures of [9, 12, 11]. We point out that the quality of the spanner could be improved to $O(\log^{1+\epsilon}(n))$ at the expense of a loss of a rather large sub-polynomial factor $n^{1/(\log \log n)^{\Theta(\epsilon)}}$ in update time and recourse using the APSP data structure of [11] (See Theorem 19). In the next section, we exploit their data structure to obtain lower stretch spanners.

5 Lower Stretch & More Deletions

The advantage of the data structure from Theorem 12 is that it has sublinear recourse if the number of insertions is much larger than the number of deletions. On the flip side, processing many deletions is expensive if the input graph is dense. In some applications, spanners are used to further sparsify graphs that are already pretty sparse [7, 5, 12, 13], but in others this may cause significant overhead.

We thus want to modify our spanner to allow for (1) an arbitrary amount of edge deletions instead of $O(n)$ as before, and (2) arbitrary stretch $\delta = \omega(1)$ instead of $n^{o(1)}$ as before. We start by noting that replacing the factor $2 \log n$ in Line 6 of Algorithm 1 with $2 \cdot \kappa$ for some arbitrary $\kappa > 0$ directly yields the following.

▶ **Lemma 13.** *Let $\kappa > 0$. Then if we replace the factor $2 \log n$ in Line 6 of Algorithm 1 by $2 \cdot \kappa$, the following properties hold for a sequence of m insertions and m/K deletions:*
1. *The graph H maintained is a $2 \cdot \gamma \cdot \kappa$ spanner throughout.*
2. *$|E(H)| \leq O(\frac{\gamma \cdot m \cdot \kappa}{K} + n^{1+\frac{1}{\kappa}})$ throughout a sequence of m insertions and m/K deletions.*
3. *The recourse of H is $O(\frac{\gamma \cdot m \cdot \kappa}{K} + n^{1+\frac{1}{\kappa}})$.*

The algorithm runs in total time $O(\beta \cdot \gamma \cdot m \cdot \kappa + \alpha \cdot \gamma \cdot \left(\frac{m}{K} \cdot \kappa + n^{1+\frac{1}{\kappa}}\right))$.

Proof. The proof of the first item is completely analogous to the proof of Lemma 6. The second item follows from Observation 1 and the bound on the number of congested edges as in Claim 7 applied analogously as in the proof of Claim 9. Finally, the last item follows from the bound on the number of edges and by the fact that edges only leave the spanner when they get deleted. ◀

5.1 Stepwise sparsification for processing more deletions

We now show that we can allow for an arbitrary amount of deletions at the cost of a subpolynomial overhead in both stretch and update time.

Previously, we chose the density reduction parameter $K = m/n$, which corresponds to completely sparsifying the graph in one step. To handle more deletions, we choose K much smaller, but then recursively compute spanners of spanners until we reach the desired sparsity. We call each such round of sparsification a layer.

Concretely, we let the density reduction K be a tunable parameter and assume $m = K^{\Lambda} \cdot n^{1+1/\kappa}$ for some integer number of layers $\Lambda = O(\log_K(m/n^{1+1/\kappa}))$. Reducing the density by K for Λ layers will ensure that we reach the desired sparsity via the algorithm outlined below.

5.2 Layered algorithm

We initialize $H_0 = G$, and recursively apply Algorithm 1 to obtain H_i from H_{i-1} for $i = 1, \ldots, \Lambda$. We let H_Λ be the desired spanner. We then dynamically maintain this hierarchy throughout a sequence of up to m insertions and m/K deletions to $H_0 = G$. Concretely, we go over the layers in ascending order and at layer i feed all the changes caused to H_{i-1} to the spanner data structure maintaining H_i. This may lead to updates of H_i, which then get passed to the next layer $i + 1$ as long as $i < \Lambda$.

Whenever the graph G has experienced some multiple of m/K^{i+1} deletions, we re-initialize layers $i, \ldots, \Lambda$. We call this a rebuild. The reason rebuilds are required is to maintain sparsity, as otherwise the number of congested edges would grow too much on layers with higher indices due to re-insertions. We stress that the timing of rebuilds only depends on the update sequence to the original graph G.

The key insight that enables this algorithm is that edge deletions in H_i always correspond to edge deletions in G, i.e. deletions to H_i might cause some extra insertions to H_{i+1}, but never more deletions. This is crucial since deletions cause a large amount of recourse, but recourse due to insertions is effectively sparsified.

We first show that the final spanner H_Λ has the desired sparsity.

▷ Claim 14 (Sparsity). At any time, $|E(H_i)| = O((\gamma \cdot \kappa)^i \cdot m/K^i)$.

Proof. We show the claim by induction on i. For $H_0 = G$, the claim follows directly. Then, for $i > 0$, the claim follows by Lemma 13 and the fact that the layers with indices $\geq i$ get re-started after an interval of m/K^{i+1} deletions happened. ◁

We bound the number of rebuilds of layer i directly.

▷ Claim 15 (Rebuilds). After m edge insertions and m/K edge deletions, the total number of rebuilds of layer i is K^i.

Proof. The claim directly follows from the description of our algorithm, i.e. a rebuild of layer i happens whenever a multiple of m/K^{i+1} deletions happened since the last rebuild. ◁

We then bound the recourse of the graph H_Λ in a more fine grained manner.

▷ Claim 16. The recourse of H_Λ after D deletions is bounded by $O((\gamma\cdot\kappa)^\Lambda\cdot n^{1+1/\kappa}+D\cdot(\gamma\cdot\kappa)^\Lambda)$.

Proof. In between two rebuilds the recourse of H_Λ is at most $O((\gamma\cdot\kappa)^\Lambda\cdot n^{1+1/\kappa})$. The total number of rebuilds is bounded by $D/n^{1+1/\kappa}$. Therefore, the claim follows. ◁

Finally, we prove that the stretch of the final spanner is sufficiently bounded as long as Λ is very small.

▷ Claim 17 (Stretch). The total stretch of the spanner computed at layer Λ with respect to G is at most $(2\cdot\gamma\cdot\kappa)^\Lambda$.

Proof. The claim follows from applying Item 1 in Lemma 13 Λ times. ◁

We conclude with the main theorem of this section.

▶ **Theorem 18.** *Given a γ-approximate α-update β-query APSP data structure (Definition 5) and parameters $\kappa > 0, 1 \le K \le m/n$, there is an algorithm that maintains a $(2\gamma\kappa)^{O(\log_K(m/n^{1+1/\kappa}))}$-approximate spanner H of an initially empty graph G on n vertices throughout an arbitrary sequence of edge insertions and deletions such that $|E(H)| = O((\gamma\cdot\kappa)^{O(\log_K(m/n^{1+1/\kappa}))}\cdot n^{1+1/\kappa})$ with amortized update time $O((\beta+\alpha)\cdot K\cdot(2\gamma\kappa)^{O(\log_K(m/n^{1+1/\kappa}))})$.*

Proof. Sparsity and stretch follow from Claim 14 and Claim 17, respectively. It remains to show the statement about run-time. Suppose we received m edge-insertions and m edge-deletions. Each layer i handles up to $(\gamma\cdot\kappa)^i m/K^i$ insertions and m/K^{i+1} deletions before being restarted, which takes time $O((\beta+\alpha)\cdot(\gamma\cdot\kappa)^i\cdot m/K^i)$ by Lemma 13. The total number of restarts of layer i is K^{i+1}, so that the total amount of time spent for layer i is $O(K^{i+1}\cdot(\beta+\alpha)\cdot(\gamma\cdot\kappa)^i\cdot m/K^i) = O(m\cdot(\beta+\alpha)\cdot(\gamma\cdot\kappa)^i\cdot K)$. Summing the costs over all Λ layers yields the statement. ◀

The following theorem from [11] allows us to instantiate our parameters in a suitable way. Since our approximation factor is larger than γ^Λ, we rely on the ability to choose a very small approximation factor γ in their APSP algorithm. Then, we choose K to be a very large sub-polynomial factor to obtain very few layers Λ.

▶ **Theorem 19** (Theorem 1.3 in [11]). *For some small constant $c > 0$, given a parameter $1/\log^c n \le \epsilon \le 1$, there is a $2^{\mathrm{poly}(1/\epsilon)}$-approximate, $O(n^\epsilon)$-update and $O(\log\log n/\epsilon^4)$-query APSP data structure.*

▶ **Theorem 20.** *Given parameters $1/\log^c n \le \epsilon \le 1$, $\kappa > 0$ and $1 \le K \le m/n^{1+1/\kappa}$, there is an algorithm that maintains a $(2^{\mathrm{poly}(1/\epsilon)}\kappa)^{O(\log_K(m/n^{1+1/\kappa}))}$-approximate spanner H of an initially empty graph G on n vertices throughout an arbitrary sequence of edge insertions and deletions such that $|E(H)| = O((2^{\mathrm{poly}(1/\epsilon)}\kappa)^{O(\log_K(m/n^{1+1/\kappa})}\cdot n^{1+1/\kappa}\cdot\kappa)$ with amortized update time $O(n^\epsilon\cdot K\cdot(2^{\mathrm{poly}(1/\epsilon)}\kappa)^{O(\log_K(m/n^{1+1/\kappa}))})$.*

Now Theorem 4 follows from the previous theorem by a specific choice of parameters.

Proof of Theorem 4. We choose $\kappa = \log \delta$, $\epsilon = 1/\log\log\delta$ and $K = n^{1/\log\log\delta}$. Note that then $\Lambda = O(\log_K(m/n^{1+1/\kappa})) = O(\log\log\delta)$.

So for this choice of parameters, the stretch is

$$(2\gamma\kappa)^\Lambda = (2^{\text{poly}(1/\epsilon)}\kappa)^\Lambda = 2^{\text{poly}(\log\log\delta)} = O(\delta).$$

The sparsity is

$$|E(H)| = O((2^{\text{poly}(1/\epsilon)}\kappa)^\Lambda \cdot n^{1+1/\kappa} \cdot \kappa) = O(2^{\text{poly}(\log\log\delta)} n^{1+1/\log\delta} \log\delta) = n^{1+o(1)},$$

and the amortized update time is

$$O(n^\epsilon \cdot K \cdot (2^{\text{poly}(1/\epsilon)}\kappa)^\Lambda) = O(n^{2/\log\log\delta} 2^{\text{poly}(\log\log\delta)}) = n^{o(1)}.$$

Finally, the desired recourse bound follows from Claim 16. ◀

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 Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4), 2012. doi:10.1145/2344422.2344425.

3 Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-Dynamic Graph Sparsifiers Against an Adaptive Adversary. In Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2022.20.

4 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.

5 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.

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 and Merav Parter. Fully dynamic algorithms for graph spanners via low-diameter router decomposition. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 785–823, 2025. doi:10.1137/1.9781611978322.23.

9 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.

10 Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, pages 377–388, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3313276.3316381.

11 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.

12 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.

13 Rasmus Kyng, Simon Meierhans, and Gernot Zöcklein. Bootstrapping dynamic apsp via sparsification, 2024. doi:10.48550/arXiv.2408.11375.