Deterministic Incremental APSP with Polylogarithmic Update Time and Stretch

Sebastian Forster
University of Salzburg
Salzburg, Austria
forster@cs.sbg.ac.at

Yasamin Nazari
University of Salzburg
Salzburg, Austria
ynazari@cs.sbg.ac.at

Maximilian Probst Gutenberg
ETH Zurich
Zurich, Switzerland
maxprobst@ethz.ch

ABSTRACT

We provide the first *deterministic* data structure that given a weighted undirected graph undergoing edge insertions, processes each update with *polylogarithmic* amortized update time and answers queries for the distance between any pair of vertices in the current graph with a *polylogarithmic* approximation in $O(\log\log n)$ time.

Prior to this work, no data structure was known for partially dynamic graphs, i.e., graphs undergoing either edge insertions or deletions, with less than $n^{o(1)}$ update time except for dense graphs, even when allowing randomization against oblivious adversaries or considering only single-source distances.

CCS CONCEPTS

• **Theory of computation → Dynamic graph algorithms**.

KEYWORDS

Dynamic algorithms, Shortest paths, Graph algorithms

ACM Reference Format:
Sebastian Forster, Yasamin Nazari, and Maximilian Probst Gutenberg. 2023. Deterministic Incremental APSP with Polylogarithmic Update Time and Stretch. 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/10.1145/3564246.3585213

1 INTRODUCTION

Partially dynamic algorithms for approximate shortest path problems have received considerable attention in recent years. In the partially dynamic setting the input graph is undergoing either edge insertions (incremental setting) or edge deletions (decreme1.265ntal setting). The focus on partially dynamic distance approximation algorithms, instead of fully dynamic ones allowing both types of updates, has three major reasons:

- Fully dynamic maintenance of exact distances or small-stretch approximations is sometimes not possible with small update time under plausible hardness assumptions [2, 36, 42, 51].
- Partially dynamic algorithms often serve as a "stepping stone" for fully dynamic algorithms [5, 30, 47].
- In several applications, partially dynamic algorithms that are deterministic or use randomization against an adaptive adversary[1] can be used as subroutines for solving static problems [15, 19–22, 41].

The research line of developing partially dynamic distance approximation algorithms against adaptive adversaries has been especially successful for *undirected* graphs: in particular, deterministic incremental and decremental algorithms with almost optimal amortized update time of $n^{o(1)}$ exist for the single-source shortest paths problem (SSSP) with stretch $(1+o(1))$ [15] and for the all-pairs shortest paths problem (APSP) with stretch $n^{o(1)}$ [21].[2] These efforts were leveraged for the following applications in static algorithms:

- $(1+o(1))$-approximate minimum-cost flow in time $m^{1+o(1)}$ [15] (using deterministic decremental $(1+o(1))$-approximate SSSP)
- $n^{o(1)}$-approximate multicommodity flow in time $m^{1+o(1)}$ [21] (using deterministic decremental $n^{o(1)}$-approximate APSP)
- Exact minimum cost flow in time $m^{1+o(1)}$ [20] (using randomized $n^{o(1)}$-approximate APSP against adaptive adversaries on expander graphs)
- Deterministic nearly-linear time constructions of light spanners (using deterministic incremental $O(1)$ -approximate bounded distance APSP).

All known algorithms for partially dynamic approximate SSSP and APSP that are deterministic (or randomized against an adaptive adversary) suffer from an "$n^{o(1)}$-bottleneck" in their update time with the only exception being the partially dynamic approximate SSSP algorithm of Bernstein and Chechik [14] that achieves polylogarithmic update time in very dense graphs with $\tilde{\Omega}(n^2)$ edges.[3] Even if we allowed randomization against an oblivious adversary, the $n^{o(1)}$-bottleneck persists for sparse graphs [40].

It is thus an intriguing and important open problem to design improved deterministic algorithms with polylogarithmic update time and polylogarithmic stretch in all density regimes that ideally work against an adaptive adversary to allow the use in static algorithms. In particular, it has recently been shown that a deterministic decremental APSP algorithm with polylogarithmic stretch and polylogarithmic update time would imply a $\tilde{O}(m)$-time algorithm for

STOC '23, June 20–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9913-5/23/06.
https://doi.org/10.1145/3564246.3585213

[1]This means that the "adversary" creating the sequence of updates is *adaptive* in the sense that it may react to the outputs of the algorithm. This type of adversary is called "adaptive online adversary" in the context of online algorithms [10]. In contrast, an *oblivious* adversary needs to choose its sequence of updates in advance, which guarantees probabilistic independence between the updates and the random choices made by the algorithm. Deterministic algorithms obviously work against an adaptive adversary.

[2]As usual, n denotes the number of vertices and m denotes the maximum number of edges of the graph. In the introductory parts of this paper, we assume that edge weights are integer and polynomial in n when stating running time bounds.

[3]In this paper, we use $\tilde{O}(\cdot)$- and $\tilde{\Omega}(\cdot)$-notation to suppress factors that are polylogarithmic in n.

finding balanced sparse cuts [19] and improve the state-of-the-art running time of the (exact) minimum cost flow problem [20].

Our Result. In this paper, we present the first shortest path algorithm breaking the $n^{o(1)}$ update-time barrier in *all density regimes* for a partially dynamic distance problem with polylogarithmic stretch. We give an incremental algorithm for maintaining a distance oracle with polylogarithmic stretch that has polylogarithmic amortized update time.

Theorem 1.1. *There is a deterministic algorithm that, given an undirected graph with real edge weights in* $[1, W]$ *undergoing edge insertions, in total time* $O(m \log n \log\log n + n \log^6(nW) \log\log n)$ *over all updates maintains a distance oracle with polylogarithmic stretch and query time* $O(\log\log n)$*, where n denotes the number of vertices m denotes the final number of edges of the graph.*

Note that, while our stretch guarantee leaves some room for improvement in the exponent of the logarithm, there is evidence that a substantial improvement might not be possible without sacrificing the polylogarithmic update time: Based on popular hardness assumptions concerning static 3SUM or static APSP, Abboud, Bringmann, Khoury, and Zamir [1] recently showed that a constant-stretch distance oracle cannot be maintained with update time $n^{o(1)}$. Furthermore, a space bound (and thus also a total update time bound) of $\Omega(n^{1+1/k})$ for distance oracles with stretch $2k - 1$ follows from Erdős's girth conjecture [27, 48].

It is worth noting that our techniques are very different from previous approaches that usually employ Even-Shiloach trees [29, 39] and constructions based on the Thorup-Zwick distance oracle [48]. We further use no heavy algorithmic machinery and – apart from a dynamic tree data structure – our paper is self-contained. We crucially use a hierarchical vertex sparsifier construction by Andoni, Stein, and Zhong [7] that originally was developed in the context of distance approximation algorithms with polylogarithmic depth for the PRAM model. The major technical challenge in employing this hierarchy in a dynamic setting is controlling the recourse – i.e., the number of induced updates – from the bottom to the top. A standard "layer-by-layer" analysis would lead to an exponential blowup that would at best result in an $n^{o(1)}$ overhead. Instead, we perform several modifications to the vertex sparsifier hierarchy that allow a more controlled propagation in the algorithm that avoids such blowups. These modifications require a more entangled analysis over different levels. Hence despite the relatively simple algorithm, our analysis is quite technical; see Section 2 for an overview of our technical ideas.

Prior Work. For the *fully dynamic* all-pairs shortest paths problem, we can in principle distinguish the two regimes of update time $\Omega(n)$ and update time $o(n)$ (the "sublinear" regime). Most earlier works have focused on the first regime [25, 39] and the state-of-the-art fully dynamic algorithms for APSP have an amortized update time of $\tilde{O}(n^2)$ to maintain an exact solution [24, 46] or an amortized update time of $m^{1+o(1)}$ to maintain a $(2 + o(1))$-approximate solution in undirected graphs [12, 15] (which can then be combined with a dynamic spanner algorithm [9]). Several approaches exist to obtain comparable worst-case update time [4, 31, 47, 49, 50] and to obtain subquadratic (but still superlinear) update time at the cost of polynomial query time [11, 38, 43–45, 51].

Work on the sublinear regime has been pioneered by Abraham, Chechik, and Talwar [5] with a trade-off between stretch and amortized update time for unweighed, undirected graphs, allowing for example for a stretch of $O(\log n)$ with amortized update time $O(n^{1/2+\delta})$ (for any constant δ) in sparse graphs with $O(n)$ edges. A trade-off for weighted, undirected graphs allowing for even faster update time has been presented by Forster, Goranci, and Henzinger [30], allowing for example for both subpolynomial stretch and subpolynomial amortized update time. Both of these algorithms are randomized and correct against an oblivious adversary.

For the *partially dynamic* all-pairs shortest paths problem, we can similarly distinguish between algorithms with total update time $\tilde{O}(mn)$ (and above) and faster "subcubic" algorithms. In the first regime, the state of the art is as follows: deterministic exact all-pairs shortest paths can be maintained with total update time $\tilde{O}(n^3)$ in unweighted, directed graphs [8, 25, 28] and $(1 + \epsilon)$-approximate all-pairs shortest paths can be maintained with total update time $\tilde{O}(mn/\epsilon)$ in weighted, directed graphs [13, 34, 44] against an oblivious adversary, and in total update time $\tilde{O}(mn^{4/3}/\epsilon^2)$ against an adaptive adversary [28, 37].

Subcubic algorithms go beyond the "mn" barrier in undirected graphs either by increasing the multiplicative stretch or by allowing extra additive stretch. In terms of purely multiplicative stretch, Chechik [17] presented an algorithm that for any integer $k \geq 2$ maintains a distance oracle of stretch $(2 + o(1))(k - 1)$ with total update time $mn^{1/k+o(1)}$, yielding in particular logarithmic stretch with total update time $m^{1+o(1)}$. This result was refined by Łącki and Nazari [40] to in particular improve the total update time to $\tilde{O}((m + n^{1+o(1)})n^{1/k})$. Prior works were relevant only for dense graphs [16] or had "exponentially growing" stretch guarantees [5, 35]. Recently, a subcubic partially dynamic algorithm with stretch $2 + o(1)$ has been developed as well [26]. All of these subcubic algorithms for multiplicative stretch are randomized and assume an oblivious adversary. A deterministic incremental algorithm with several trade-offs between stretch and update and query time was developed by [18]; in particular their algorithm can provide constant stretch and total update time $m^{1+o(1)}$. Deterministic partially dynamic algorithms allowing deletions have been developed by Chuzhoy and Saranurak [23] and by Chuzhoy [21], where the latter work provides polylogarithmic stretch in total update time $O(m^{1+\delta})$ for any constant δ (see also [15]). The state-of-the-art algorithm with "mixed" stretch guarantee has a multiplicative stretch of $(1 + o(1))$, an additive stretch of $2(k - 1)$, and a total update time of $(n^{2-1/k+o(1)}m^{1/k}$ [26]. Prior works considered only the case $k = 2$ [3, 34] or had an "exponentially growing" stretch guarantee [33]. Again, all of these subcubic algorithms for "mixed" stretch are randomized and assume an oblivious adversary

2 OVERVIEW

We start by reviewing the techniques from [7]. We then explain several challenges we face in the dynamic settings and the modifications we make to the construction to overcome these challenges.

Review of the static construction of [7]. Our starting point is a distance oracle proposed by [7] that supports fast distance queries with polylogarithmic stretch. We can see this structure as a *hierarchy of vertex sparsifiers*[4]. For a given graph $G = (V, E)$, a vertex sparsifier is a graph H, where $V(H) \subseteq V(G)$ and each vertex $v \in V$ has a representative vertex $p(v)$ (called a pivot) such that the distance between vertices $u, v \in V$ can be approximated with the distance between $p(u)$ and $p(v)$ in H. A hierarchy of vertex sparsifiers allows us to compute approximate distances on subsequently smaller graphs in each level, while trading off computation time with the stretch.

Specifically, the algorithm of [7] creates graphs $H_1, \ldots, H_k$ in $k = O(\log \log n)$ levels as follows: at each level i for a parameter b_i, they choose $V(H_{i+1}) \subseteq V(H_i)$ by subsampling a set of $\tilde{O}(|V(H_i)|/b_i)$ vertices. By choosing appropriate b_i values increasing double exponentially in each level (in our case by setting $b_i = 2^{(6/5)^i}$) we have that after $k = O(\log \log n)$ levels the number of remaining vertices is very small. Roughly speaking, each level of the hierarchy incurs an additional constant multiplicative factor in the stretch, which we denote by α and it then remains to observe that $\alpha^k = $ polylog n. We next describe the procedure by [7] to compute a vertex sparsifier H' for a graph H with some target parameter b such that $|V(H')| \approx |V(H)|/b$. The hierarchy $H_1, \ldots, H_k$ is then obtained recursively by computing H_{i+1} from H_i by using the described procedure with target parameter b_i.

The procedure initially samples each vertex from $V(H)$ with probability $1/b$ to form the vertex set of H'. They define the pivot $p(v)$ for any vertex $v \in V(H)$ to be the vertex in $V(H') \subseteq V(H)$ that is closest to v in H (we assume that distances are unique for simplicity in the overview). We define $pivotDist(v)$ to be the distance from v to its pivot in H. Using standard hitting set arguments, one can then argue that there are at most $\tilde{O}(b)$ vertices inside the ball $B_H(v, pivotDist(v))$ centered at v with radius $pivotDist(v)$. Having found the vertex set of H', it remains to define the edge set. Two types of edges are added[5] to H':

- Type 1 (Ball edges): For each $v \in V(H), u \in B_H(v, pivotDist(v))$, we have an edge $(p(u), p(v)) \in H'$ of weight $pivotDist(v) + dist_H(u, v) + pivotDist(u)$, and
- Type 2 (Projected edges): For each $e = (x, y) \in E(H)$, an edge $(p(x), p(y))$ of weight $pivotDist(x) + w_H(x, y) + pivotDist(y)$ is added to H'.

Intuitively, the first type of edges connect vertices in H if one vertex appears in the ball of the other and the second type of edges connect the boundaries of the balls.

We briefly sketch the stretch analysis: Consider vertices $u, v \in V(H)$, and let π be the shortest path between u and v in H. We divide this path into segments defined by a sequence of vertices $u = y_0, ..., y_{\ell-1}$ and $x_1, ..., x_\ell = v$ defined as follows: Starting from $y_0 := u$, for each $s > 0$ let x_{s+1} be the last vertex on π such that $x_{s+1} \in B_H(y_s, pivotDist(y_s))$. Then y_{s+1} is set to the next vertex on π right after x_{s+1}. We stop when $x_{s+1} = v$. Essentially, we segment the path π to alternately take maximal segments contained in balls to pivot distances, and using the original edge (see also Figure 1).

[4]In [7] what we call a vertex sparsifier is called a sub-emulator. They use a type of sub-emulator for building a low-hop emulator, i.e. a graph that approximates the distances only using paths with $O(\log \log n)$-hops. We do not need a low-hop emulator and instead use subemulators/vertex sparsifiers for maintaining a distance oracle with small update and query time.

[5]Based on this definition, we may be introducing multi-edges to H'.

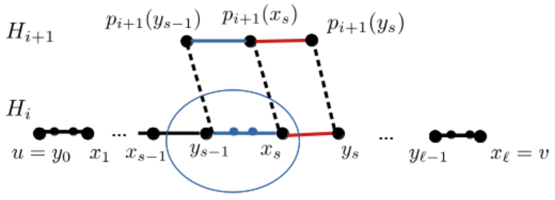

Figure 1: In the construction of [7] paths in H_i are approximated by paths in H_{i+1} going through level i pivots (denoted by the function p_{i+1}) of certain vertices on the path. Here x_s is in the ball of y_{s-1} (represented by the circle). Dashed lines represent projections to the next level.

[7] then suggest that this path can simply be projected to H' by taking the path $\langle p(u = y_0), p(x_1), p(y_1), p(x_2), \ldots, p(x_\ell = v)\rangle$ in H' where every two vertices are connected by an edge in H' as can be verified from the procedure above.

One can then show that $dist_{H'}(p(y_s), p(y_{s+1})) \approx dist_H(y_s, y_{s+1}) + pivotDist(y_s) + pivotDist(y_{s+1})$ (here $\approx$ hides a constant factor). But it is not hard to see that for any s, we have $dist_H(y_s, y_{s+1}) > pivotDist(y_s)$. Summing over path segments, we thus get that $dist_{H'}(p(u), p(v)) \approx dist_H(u, v) + pivotDist(v)$.

Unfortunately, one cannot hope to get rid of an additive term scaling linearly in $pivotDist(v)$ in the approximation as can be seen from straight-forward worst-case examples. However, one can use classic distance oracle query techniques: For query pair u, v, we either have that $u \in B_H(v, pivotDist(v))$ and keeping these balls and the respective distances explicitly in a dictionary, one can then return the exact distance. Otherwise, we have $pivotDist(v) < dist_H(u, v)$ and therefore $dist_{H'}(p(u), p(v)) \approx dist_H(u, v)$.

Finally, it is easy to see that the sampling ensures $|E(H')| \leq \tilde{O}(|V(H)|b) + |E(H)|$.

Putting this result back to our hierarchy $H_1, H_2, \ldots, H_k$, we have that one can straight-forwardly query the distances between any two vertices in the original graph G in time $O(k)$ by applying the discussed query procedure iteratively. Further, for each i, we have $|E(H_i)| \leq |E(G)| + \sum_i O(|V(H_i)|/b_i)$ where b_i's are chosen carefully to ensure $|E(H_i)| \leq m + \tilde{O}(n)$.

Algorithm 1: UpdateApproxPivots(H, b)

```
1 while $\exists v \in V(H)$ such that $|B_H(v, \frac{1}{4}\widetilde{pivotDist}(v))| \geq b$ do
2     Let $B_v$ be a set of size $b$ such that
      $B_v \subseteq B_H(v, \frac{1}{4}\widetilde{pivotDist}(v))$.
3     if $\exists u \in B_v$ with $\widetilde{pivotDist}(u) < \frac{1}{2}\widetilde{pivotDist}(v)$ then
4         $p(v) \leftarrow p(u)$;
          $\widetilde{pivotDist}(v) \leftarrow dist_H(v, u) + \widetilde{pivotDist}(u)$.
5     else
6         foreach $u \in B_v$ do $p(u) \leftarrow v$;
              $\widetilde{pivotDist}(u) \leftarrow dist_H(u, v)$.
```

Incremental algorithm for one level. We give an overview of our algorithm for maintaining these two types of edges for a single level before describing the modifications needed for making the algorithm efficient over all levels.

The first obstruction to maintaining the vertex sparsifier of the last section in incremental settings is that for each vertex $v \in V(H)$, even if the pivot $p(v)$ does not change, the pivot distance $pivotDist(v)$ might change after almost each of the m insertions. While on average most vertices might only undergo few pivot distance changes, we still might have some node v of large degree and would have to adjust the weight of projected edges (type 2) incident on v with every change in v's pivot distance. To avoid such a running time overhead for simply maintaining the weights of projected edges, we maintain an approximation $\widetilde{pivotDist}(v)$ of $pivotDist(v)$ such that whenever $\widetilde{pivotDist}(v)$ changes, it decreases by a constant factor (thus the total number of changes is $O(\log(nW))$. We maintain the pivot $p(v)$ to be some vertex in $V(H')$ that is roughly at distance $\widetilde{pivotDist}(v)$ (in our case, all approximations are within a factor 4 of each other). In the following, we merely describe an algorithm to maintain the approximate pivots and the corresponding distance estimates as it is straight-forward to maintain the edge set of H' from this information.

We give our procedure in Algorithm 1. We assume here that all edge weights in H are powers of 2 which is w.l.o.g. since we only want to obtain constant stretch. Here, we skip the initialization procedure for brevity. The algorithm is then invoked after every edge insertion to H. The algorithm works as follows: whenever it detects that the ball $B_H(v, \frac{1}{4}\widetilde{pivotDist}(v))$ contains more than b vertices, it checks for a closer pivot for v which then decreases $\widetilde{pivotDist}(v)$ significantly. Therefore, the algorithm searches over b vertices in $B_H(v, \frac{1}{4}\widetilde{pivotDist}(v))$ and asks them whether their pivot would make a good candidate. If such a candidate is found, it becomes the new pivot of v. Otherwise, we make v a pivot itself and assign it to the set of vertices scanned as a pivot (the vertices in B_v). In this latter case, each vertex in B_v has its (approximate) pivot distance decreased significantly. We maintain the vertex set $V(H')$ to be the image of all pivot functions p from the current and previous stages.

To implement the while-loop in Algorithm 1, we use a truncated Dijkstra's algorithm from each vertex v to explore $B_H(v, \frac{1}{4}\widetilde{pivotDist}(v))$, however, we abort the procedure after seeing b vertices. Using adjacency lists sorted by weight, we can implement this procedure in time $\tilde{O}(b^2)$. Note that in between any two stages, for a vertex v, if no edge/a multi-edge of equal or higher weight is inserted into $B_H(v, \frac{1}{4}\widetilde{pivotDist}(v))$, we can simply ignore the update and do not need to recompute. But in the other case, for fixed $\widetilde{pivotDist}(v)$, we have that $B_H(v, \frac{1}{4}\widetilde{pivotDist}(v))$ is increasing over time, and the number of edges with different weights and endpoints in the final ball (before exceeding b vertices) is $b^2 \log(nW)$ edges. Hence we have that there are at most $O(b^2 \log nW)$ recomputations before $\widetilde{pivotDist}(v)$ changes. Overall this incurs total time $\tilde{O}(b^4 \log^2 nW)$ per vertex $v \in V(H)$, and thus $\tilde{O}(m + |V(H)|b^4 \operatorname{polylog} nW)$ overall.[6]

[6]Here, we were slightly imprecise but in the exact analysis one has a higher power for the $\log(nW)$ factor.

Finally, observe that $|V(H')|$ is only increased if we enter the else-case. But in this case, b vertices have their pivot distance significantly decreased. We can therefore upper bound the number of vertices in H' by $O(|V(H)|\log(nW)/b)$. However as we see next, bounding the recourse on the number of edges will be problematic over all the k levels.

Challenges in maintaining Projected Edges. The naive approach for maintaining the vertex-sparsifier hierarchy would be to run the aforementioned algorithm for each $1 \leq i < k$ in a black-box manner to maintain H_{i+1} as a vertex sparsifier of H_i. In particular, the edges added to H_{i+1} over the course of the algorithm appear as insertions to the algorithm maintaining the vertex sparsifier H_{i+2} of the next level. Thus $O(|E(H_i)| \log nW + |V(H_i)| b_i^2 \log(nW))$ edges are inserted to H_{i+1} in total: $O(|V(H_i)| b_i^2 \log(nW))$ type 1 (ball) edges, and $O(|E(H_i)| \log nW)$ type 2 (projected) edges.

Starting from $G = H_1$ with m edges, this naive approach with $k = O(\log\log n)$ levels leads to a bound of at least $m \cdot O(\log nW)^{\log\log n}$ type 2 edges inserted to the top level, each of which needs *at least* constant time to be processed. Therefore, the black-box approach will not give us the desired $\tilde{O}(m)$ total update time. Instead, we propose a more careful approach for avoiding the exponential blow up in the number of inserted edges within the hierarchy that we explain next.

Maintaining the hierarchy via multi-level projections. The challenge discussed means that we cannot afford to have a chain of projections from lower levels to higher levels. In the following, $p_{i+1}(v)$ denotes the pivot of some node $v \in V(H_i)$ maintained by the incremental algorithm at level i and $\widetilde{pivotDist}_{i+1}(v)$ denotes its corresponding approximate pivot distance. For the sake of concreteness, consider some edge (u, v) in $H_1 = G$ and the pivots $p_2(u) \in V(H_2)$ and $p_2(v) \in V(H_2)$ of its endpoints. Following our previous definition of H_2 and our process for maintaining approximate pivot distances, there would be a projected edge $(p_2(u), p_2(v))$ in H_2 of weight:

$$w_{H_2}(p_2(u), p_2(v)) = \widetilde{pivotDist}_2(u) + w_{H_1}(u, v) + \widetilde{pivotDist}_2(v) .$$

This edge is projected to H_3 by edge $(p_3(p_2(u)), p_3(p_2(v)))$ of weight

$$\begin{aligned} &w_{H_3}(p_3(p_2(u)), p_3(p_2(v))) = \\ &\quad \widetilde{pivotDist}_3(p_2(u)) + w_{H_2}(p_2(u), p_2(v)) \\ &\quad + \widetilde{pivotDist}_3(p_2(v)) = \widetilde{pivotDist}_3(p_2(u)) + \widetilde{pivotDist}_2(u) \\ &\quad + w_{H_1}(u, v) + \widetilde{pivotDist}_2(v) + \widetilde{pivotDist}_3(p_2(v)) . \end{aligned}$$

As explained above, the "black box" approach would mean to insert projections of (u, v) to H_3 whenever $p_3(p_2(u))$ changes, which happens $O(\log(nW))$ times for each of the $O(\log(nW))$ choices of $p_2(u)$. This bounds the number of insertions of projections of (u, v) to H_3 by $O(\log(nW)^2)$ (and in general the number of insertions to H_i by $O(\log(nW))^{i-1}$).

Our main idea for obtaining a better bound is to employ another lazy updating scheme: we insert a projection of (u, v) to H_3 only when the sum above determining the edge weight changes significantly, in particular whenever the "left part" $\widetilde{pivotDist}_3(p_2(u)) + \widetilde{pivotDist}_2(u)$ or the "right part" $\widetilde{pivotDist}_2(v) + \widetilde{pivotDist}_3(p_2(v))$

decreases by a constant factor. In this way, we "reproject" (u, v) to H_3 only $O(\log(nW))$ times, a bound that is independent on the level at which the projection happens, which gives us the desired control in the number of insertions at each level. Note that such projections to higher levels are not only carried out for the edges of G, but also for the type 1 (ball) edges introduced at each level of the hierarchy, which can be done analogously.

More precisely, we define a set of *base edges*, which are intuitively the level i edges that were not previously projected from a lower level. To this end, it is convenient to define $p_i(u) = p_i(\ldots p_2(p_1(u))\ldots)$ for any vertex $u \in V$. Then at level $i+1$, we add, in the lazy fashion explained above, a *projected edge* $(p_{i+1}(u), p_{i+1}(v)) \in H_{i+1}$ corresponding to each base edge $(u, v) \in E(H_j)$ from level $j \leq i$ and setting the weight (at time of projection) to be $\sum_{j\leq i} \widetilde{pivotDist}_j(u) + w_{H_j}(u, v) + \sum_{j\leq i} \widetilde{pivotDist}_j(v)$.

We show that we can carry out our idea efficiently by utilizing a dynamic tree data structure on the forest induced by connecting each vertex of the hierarchy to its pivot at the next level (weighted by approximate pivot distance). Whenever for some vertex v in some H_j the sum of the approximate pivot distances along the tree path to its ancestor pivot v' at some level $i > j$ decreases by a constant factor, we insert to H_i the projections of all (non-projected) edges incident on v in H_j. We call the corresponding pivot v' a significantly improving pivot of v at level i. These significantly improving pivots will play a major role in our algorithm, as we explain next.

Challenges introduced by considering significantly improving pivots. While we have abandoned the "black-box" level-by-level approach for efficiency reasons, we still want to, in spirit, follow the proof strategy of [7], which is an inductive level-by-level stretch analysis. This "mismatch" causes certain issues. Consider again the argument of [7] to show that any shortest path π in H_i has a suitable approximation in H_{i+1} (see Figure 1). The path π is divided into segments and each segment is represented by an edge in H_{i+1}. In particular, some of these segments consist of single edges (x_s, y_s), which in particular are type 2 edges in H_i. In the original proof, H_{i+1} contains the projection $(p_{i+1}(x_s), p_{i+1}(y_s))$ of (x_s, y_s) (where $p_{i+1}(x_s)$ and $p_{i+1}(y_s)$ are the current pivots of x_s and y_s, respectively).

However, after our modifications for lazy updating we only have the weaker guarantee that (x_s, y_s) was inserted previously as the projection of some (non-projected) edge $(\bar{x}_s, \bar{y}_s)$ from some lower level $j < i$. Additionally we know that H_{i+1} contains the projection e of the edge $(\bar{x}_s, \bar{y}_s)$ from H_j and the endpoints of e are the the last significantly improving pivots at level $i+1$ of $\bar{x}_s$ and $\bar{y}_s$, respectively (see Figure 2 in Section 4.1). The major challenge now is to still find a suitable path from $p_{i+1}(x_s)$ to $p_{i+1}(y_s)$ in H_{i+1}, which should include e to somehow relate the length of this path to the weight of (x_s, y_s).

New edges for significantly improving pivots. We address this challenge by introducing two new types of edges (with appropriately chosen weights) into our vertex sparsifiers: The first new type gives us an edge from the current pivot of x_s (i.e., $p_{i+1}(x_s)$) to the last significantly improving pivot of x_s. The second new type gives us an edge from the last significantly improving pivot of x_s to the last significantly improving pivot of $\bar{x}_s$, i.e., the first endpoint of e. Similarly, we can use the new types of edges to find a path from the second endpoint of e to the current pivot of y_s (i.e., $p_{i+1}(y_s)$), and thus find the desired path from $p_{i+1}(x_s)$ to $p_{i+1}(y_s)$. Since the new types of edges are used in a somewhat special configuration, we can argue that they can be included in the hierarchy with only polylogarithmic overheads. Setting the edge weights appropriately to obtain a stretch bound for this path in H_{i+1} requires some intricate estimates. The exact definition of these edges and the full analysis can be found in Section 4.1.

3 PRELIMINARIES

Basic Notation. For a general (multi-)graph H, we denote the edge set of the graph by $E(H)$, its vertex set by $V(H)$ and its weight function by w_H where w_H maps each edge in $E(H)$ to a positive number. We denote the distance between any two vertices $u, v \in V(H)$ in the graph H by $dist_H(u, v)$. We denote by $B_H(u, r) = \{dist_H(u, v) \leq r\}$ the ball at u in H of radius r. We say that H is incremental if it is undergoing edge insertions.

In this article, we denote by $G = (V, E, w)$ the input graph and define $n := |V|$, $m = |E|$ and let w be the weight function with image in $[1, W]$.

Encoding of the Adjacency List. We assume that additional to the usual encoding, we have for each (multi-)graph H an adjacency list for each vertex v denoted by $\textsc{Adj}_{H,v}$ stored as a doubly-linked list where the edges incident to v appear sorted lexicographically first by weights and then by time of arrival. Here we define time of arrival for an edge to be equal to the number of edges that were in the graph before the edge was added where we assume without loss of generality that edges are added one after another and the initial graph H is empty. We often index the adjacency list like an array and use $\textsc{Adj}_{H,v}[1, b]$ to refer to the set of the first b edges in the adjacency list of v (i.e. the b edges of smallest weight).

Update time. The *total update time* of an incremental algorithm is (a bound on) the sum of the running times spent by the algorithm for processing all of the m insertions and its *amortized update time* is its total update time divided by m.[7]

Miscellaneous. We define $\lceil x\rceil_2 = \lceil x/2\rceil \cdot 2$, where we round up x to the next multiple of 2.

We refer to the t-th *stage* of a dynamic algorithm as the instructions it performs after the t-th update. We refer to the value of a variable or function at stage t as the value directly after the t-th stage and write it with the superscript "(t)" ; $p^{(t)}(v)$ for example denotes the pivot of v at stage t. We omit the superscript when it is clear from the context, for instance when we talk about the current stage.

4 FULL ALGORITHM AND ANALYSIS

We start by giving the hierarchy that we maintain. We then give an algorithm to maintain the hierarchy efficiently that allows for additional query access. Finally, we give the query algorithm.

[7]Similarly, the total update time of a decremental algorithm is usually the sum of the running times spent by the algorithm for processing up to m deletions in a graph with initially m edges.

4.1 A Distance-Preserving Vertex Sparsifier Hierarchy

Definition 4.1 (Distance-Preserving Vertex Sparsifier Hierarchy)**.** Given an incremental, undirected, weighted graph $G = (V, E, w)$, a k-level hierarchy maintaining algorithm is an algorithm that maintains vertex sparsifiers $H_1, H_2, \dots, H_k$ for some positive integer k, with $V(H_1) \supseteq V(H_2) \supseteq \dots \supseteq V(H_k) \neq \emptyset$ where $H_1 = G$ and for every $1 \le i \le k$, H_i is an *incremental* graph (with vertex insertions). We have a pivot function set to $p_1(v) = v$ for the initial level. The algorithm maintains for every $1 \le i < k$:

(1) an approximate pivot function $p_{i+1} : V \mapsto V(H_{i+1})$ that acts as the identity on $V(H_{i+1})$, and an estimator of the distance from each $v \in V(H_i)$ to its approximate pivot $\widetilde{pivotDist}_{i+1}(v)$. We enforce that

$$dist_{H_i}(v, V(H_{i+1})) \le dist_{H_i}(v, p_{i+1}(v)) \le \widetilde{pivotDist}_{i+1}(v) \le 4 \cdot dist_{H_i}(v, V(H_{i+1})).$$

For each $v \in V$, we also maintain an estimate $\widetilde{pivotDist}_{i+1}(v) = \sum_{j \le i} \widetilde{pivotDist}_{j+1}(p_j(v))$ and the value $\widetilde{minPivotDist}_{i+1}(v) = \min_{t' \le t} \widetilde{pivotDist}^{(t')}_{i+1}(v)$ where t is the current stage of the graph. For each $v \in V \setminus V(H_{i+1})$, we have $p_{i+1}(v) = p_{i+1}(p_i(v))$.

(2) It further maintains for each $v \in V$, the *last (significantly) improving pivot* $\overline{p}_{i+1}(v)$ that we define to be the approximate pivot $p^{(t')}_{i+1}(v)$ for $t' = \min\{t'' \mid \widetilde{pivotDist}^{(t'')}_{i+1}(v) \le \lceil \widetilde{minPivotDist}_{i+1}(v) \rceil_2\}$.

Given these values, our algorithm maintains each H_{i+1} as an incremental graph consisting of two types of edges in $E(V_{i+1})$: *base edges* E^{base}_{i+1} which are the edges first introduced in level $i+1$, and *projected edges* E^{proj}_{i+1} which are projected to level $i+1$ from lower level graphs. For convenience, we define the sets $E^{base}_1 = E$ and $E^{proj}_1 = \emptyset$.
The algorithm is required to maintain a set of base edges E^{base}_{i+1} which contains

(3) for each $u \in V(H_i)$, and $v \in B_{H_i}(u, \frac{1}{4} \cdot \widetilde{pivotDist}_{i+1}(u))$, edge $(p_{i+1}(u), p_{i+1}(v))$ in E^{base}_{i+1} with weight $8 \cdot \lceil \widetilde{pivotDist}_{i+1}(u) \rceil_2$.

(4) For any vertex $v \in V(H_i)$, let $0 = t_1 < t_2 < \dots < t_h \le t$ be such that for $j \ge 1$, we have t_{j+1} to be the first stage after stage t_j such that $p^{(t_{j+1})}_{i+1}(v) \neq p^{(t_{j+1}-1)}_{i+1}(v)$. Then, we have for any $1 \le j < \ell \le h$, a base edge $(p^{(t_j)}_{i+1}(v), p^{(t_\ell)}_{i+1}(v)) \in E^{base}_{i+1}$ with weight $8 \cdot \lceil \widetilde{minPivotDist}^{(t_j)}_{i+1}(v) \rceil_2$.

(5) For any vertex $v \in V$, times $t' \le t$, we have at stage t, for $x = \overline{p}^{(t')}_i(v)$, an edge $(\overline{p}^{(t)}_{i+1}(x), \overline{p}^{(t)}_{i+1}(v))$ in E^{base}_{i+1} of weight $\lceil \widetilde{minPivotDist}^{(t)}_{i+1}(x) \rceil_2 + \lceil \widetilde{minPivotDist}^{(t')}_i(v) \rceil_2 + \lceil \widetilde{minPivotDist}^{(t)}_{i+1}(v) \rceil_2$.

Additionally, the algorithm maintains a set of projected edges E^{proj}_{i+1} which contains

(6) for $j \le i$ and $e = (x, y) \in E^{base}_j$, the edge $(\overline{p}_{i+1}(x), \overline{p}_{i+1}(y))$ in E^{proj}_{i+1} with weight $\lceil \widetilde{minPivotDist}_{i+1}(x) \rceil_2 + \lceil w_{H_j}(e) \rceil_2 + \lceil \widetilde{minPivotDist}_{i+1}(y) \rceil_2$.

We also set for all $v \in V$ $\bar{p}_1(v) = p_1(v) = v$.

We first establish the following simple facts that prove useful in the next proof of the main theorem of this section.

Fact 4.2. *For any $v \in V(H_i)$, we have $dist_{H_i}(v, p_{i+1}(v)) \le 4 \cdot \widetilde{minPivotDist}_{i+1}(v)$.*

Proof. At any stage t, let $t' = \mathrm{argmin}_{t'' \le t} \widetilde{pivotDist}^{(t'')}_{i+1}(v)$, and note that $\widetilde{minPivotDist}_{i+1}(v) = \min_{t'' \le t} \widetilde{pivotDist}^{(t'')}_{i+1}(v)$. Using that $v \in V(H_i)$, Item 1 and that H_i is incremental, we have that

$$\begin{aligned} dist_{H_i}(v, p_{i+1}(v)) &\le 4 \cdot dist_{H_i}(v, V(H_{i+1})) \\ &\le 4 \cdot dist_{H_i^{(t')}}(v, V(H^{(t')}_{i+1})) \\ &\le 4 \cdot \widetilde{pivotDist}^{(t')}_{i+1}(v) \le 4 \cdot \widetilde{minPivotDist}_{i+1}(v). \end{aligned}$$

□

We can now prove the main result of this section: we show that any algorithm that maintains a hierarchy $H_1, H_2, \dots, H_k$ as described in Definition 4.1 has distances in H_{i+1} being constant-factor approximations of distances in H_i.

Theorem 4.3. *Given a k-level hierarchy maintaining algorithm as described in Definition 4.1, for any $1 \le i \le k-1$, we have for any $u, v \in V(H_{i+1})$, that $dist_G(u, v) \le dist_{H_{i+1}}(u, v) \le 3629 \cdot dist_{H_i}(u, v)$.*

We defer the proof of the lower bound to the full version, and focus for the rest of the section on achieving the upper bound.

Creating Path Segments. Let π be the shortest path between u and v in H_i. We show that there is a path with the desired stretch in H_{i+1} by dividing this path into segments defined by a sequence of vertices $u = y_0, ..., y_{\ell-1}$ and $x_1, ..., x_\ell = v$ on π found by the following procedure:

- $y_0 \leftarrow u$, $s \leftarrow 0$, repeat the following two steps:
- Let x_{s+1} be the last vertex on π such that $dist_{H_i}(y_s, x_{s+1}) \le \frac{1}{4} \widetilde{pivotDist}_{i+1}(y_s)$. If $x_{s+1} = v$, the procedure terminates.
- Otherwise, find y_{s+1} to be the vertex that appears next on π after x_{s+1}. $s \leftarrow s + 1$.

Mapping Path Segments into H_{i+1}. For $s = 0, \dots, \ell-1$, we have by Item 3 that edge $(p_{i+1}(y_s), p_{i+1}(x_{s+1}))$ exists in H_{i+1} of weight $8 \cdot \lceil \widetilde{pivotDist}_{i+1}(y_s) \rceil_2$. For $s = 1, \dots, \ell-1$, we need to find paths from $p_{i+1}(x_s)$ to $p_{i+1}(y_s)$. However, this turns out to be a considerably more laborious task. The following lemma summarizes the result.

Lemma 4.4. *For $s = 1, \dots, \ell-1$, we have*

$$dist_{H_{i+1}}(p_{i+1}(x_s), p_{i+1}(y_s)) \le 69 \cdot w_{H_i}(x_s, y_s) + \sum_{z \in \{x_s, y_s\}} 85 \cdot \lceil \widetilde{minPivotDist}_{i+1}(z) \rceil_2.$$

Proof. Since x_s and y_s are neighbors in H_i we either have $(x_s, y_s) \in E^{base}_i$ or $(x_s, y_s) \in E^{proj}_i$. In the following argument both we handle both of these cases simultaneously. By construction of the

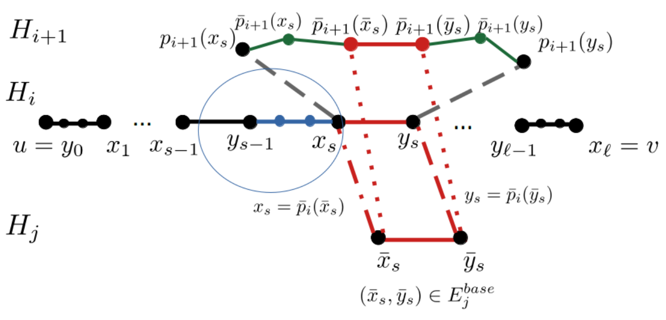

Figure 2: A sketch of the paths discussed in Lemma 4.4 and the projections from level i to $i+1$ and also from a level $j < i$ to $i+1$. Note that here in applying Claim 4.5 we set $z = \{x_s, y_s\}$. Dashed lines represent projections from lower levels to higher levels. Here x_s is in the ball of y_{s-1} and we have $(x_s, y_s) \in H_i$.

path segments, and by Item 6, that there is an edge $(\overline{x_s}, \overline{y_s}) \in E_j^{base}$ for some $j \le i$, such that $(\overline{p}_i^{(t')}(\overline{x_s}), \overline{p}_i^{(t')}(\overline{y_s})) = (x_s, y_s)$ at some stage $t' \le t$. Note that here if $(x_s, y_s) \in E_i^{base}$ then we are using the fact that $\bar{x}_s = x_s$ and $\bar{y}_s = y_s$.

Again by Item 6, we have $(\overline{p}_{i+1}(\overline{x_s}), \overline{p}_{i+1}(\overline{y_s})) \in E_{i+1}^{proj}$ of weight $\lceil \widetilde{minPivotDist}_{i+1}(\overline{x_s}) \rceil_2 + \lceil w_{H_j}(\overline{x_s}, \overline{y_s}) \rceil_2 + \lceil \widetilde{minPivotDist}_{i+1}(\overline{y_s}) \rceil_2$.

It remains to find paths from $p_{i+1}(x_s)$ to $\overline{p}_{i+1}(\overline{x_s})$ and from $p_{i+1}(\overline{y_s})$ to $p_{i+1}(y_s)$. To this end, we employ the simple claim below. This establishes the existence of a path $p_{i+1}(x_s) \rightsquigarrow \overline{p}_{i+1}(\overline{x_s}) \rightsquigarrow \overline{p}_{i+1}(\overline{y_s}) \rightsquigarrow p_{i+1}(y_s)$.

Claim 4.5. *For any $v \in V$ and $t' \le t$, where we define $z = \overline{p}_i^{(t')}(v)$,*

$$dist_{H_{i+1}}(p_{i+1}^{(t)}(z), \overline{p}_{i+1}^{(t)}(v)) \le \lceil \widetilde{minPivotDist}_i^{(t')}(v) \rceil_2 + 17 \lceil \widetilde{minPivotDist}_{i+1}^{(t)}(z) \rceil_2 + \lceil \widetilde{minPivotDist}_{i+1}^{(t)}(v) \rceil_2$$

.

Proof. We show that the path $\langle p_{i+1}(z), \overline{p}_{i+1}(z), \overline{p}_{i+1}(v) \rangle$ exists in H_{i+1} and is of small weight. Note that it is possible for some of these vertices to be the same, e.g. $p_{i+1}(z) = \overline{p}_{i+1}(z)$, but this would be a simpler case, as nothing needs to be show for the corresponding edge. Otherwise, we show the existence of each edge on this path one-by-one:

Edge $(p_{i+1}(z), \overline{p}_{i+1}(z))$: note that since $z \in V(H_i)$, we have that $\overline{p}_{i+1}^{(t)}(z) = p_{i+1}^{(t'')}(z)$ for some t'' by the definition of last improving pivots (see Item 2). But note that by Item 4, we thus have an edge $(p_{i+1}^{(t'')}(z), p_{i+1}^{(t)}(z))$ of weight $8 \cdot \lceil \widetilde{minPivotDist}_{i+1}^{(t'')}(z) \rceil_2$. Using that $\overline{p}_{i+1}^{(t)}(z)$ is updated by $p_{i+1}^{(t)}(z)$ whenever $\widetilde{minPivotDist}_{i+1}(z)$ improves by at most a factor of two, we can further upper bound the edge weight by $16 \cdot \lceil \widetilde{minPivotDist}_{i+1}^{(t)}(z) \rceil_2$.

Edge $(\overline{p}_{i+1}(z), \overline{p}_{i+1}(v))$: by Item 5 this edge is in E_{i+1}^{base} with weight

$$\lceil \widetilde{minPivotDist}_{i+1}^{(t)}(z) \rceil_2 + \lceil \widetilde{minPivotDist}_i^{(t')}(v) \rceil_2 + \lceil \widetilde{minPivotDist}_{i+1}^{(t)}(v) \rceil_2.$$

The distance then follows by summing over the upper bounds on the edge weights. □

Straight-forward addition of the upper bounds on the path segments of the exposed path $p_{i+1}(x_s) \rightsquigarrow \overline{p}_{i+1}(\overline{x_s}) \rightsquigarrow \overline{p}_{i+1}(\overline{y_s}) \rightsquigarrow p_{i+1}(y_s)$ thus establishes that

$$\begin{aligned} & dist_{H_{i+1}}(p_{i+1}(x_s), p_{i+1}(y_s)) \\ & \le \sum_{z \in \{x_s, y_s, \overline{x_s}, \overline{y_s}\}} 17 \cdot \lceil \widetilde{minPivotDist}_{i+1}(z) \rceil_2 \\ & + \lceil w_{H_j}(\overline{x_s}, \overline{y_s}) \rceil_2 + \sum_{z \in \{\overline{x_s}, \overline{y_s}\}} \lceil \widetilde{minPivotDist}_i^{(t')}(z) \rceil_2. \end{aligned}$$

To simplify this expression, we note that $w_{H_i}(x_s, y_s)$ is

$$= \lceil \widetilde{minPivotDist}_i^{(t')}(\overline{x_s}) \rceil_2 + w_{H_j}(\overline{x_s}, \overline{y_s}) + \lceil \widetilde{minPivotDist}_i^{(t')}(\overline{y_s}) \rceil_2.$$

For $j < i$, this follows from Item 6. For $j = i$, by Item 1 we have $x_s = \overline{x_s}$, $y_s = \overline{y_s}$ and thus

$$\widetilde{minPivotDist}_i^{(t')}(\overline{x}) = \widetilde{minPivotDist}_i^{(t')}(\overline{x}) = 0.$$

Further, for $z \in \{x_s, y_s\}$, using first Item 1 and then Fact 4.2 yields

$$\begin{aligned} \widetilde{minPivotDist}_{i+1}(\overline{z}) &= \sum_{j \le j' \le i} \widetilde{pivotDist}_{j'+1}(p_{j'}(\overline{z})) \\ & \le 4 \cdot \sum_{j \le j' \le i} dist_{H_{j'}}(p_{j'}(\overline{z}), V(H_{j'+1})) \\ & \le 4 \cdot \left(\lceil \widetilde{minPivotDist}_i^{(t')}(\overline{z}) \rceil_2 + dist_{H_i}(z, p_{i+1}(z)) \right) \end{aligned}$$

and therefore

$$\begin{aligned} & \widetilde{minPivotDist}_{i+1}(\overline{x_s}) + \widetilde{minPivotDist}_{i+1}(\overline{y_s}) \\ & \qquad \le 4 \left(w_{H_i}(x_s, y_s) + \sum_{z \in \{x_s, y_s\}} \widetilde{minPivotDist}_{i+1}(z) \right). \end{aligned}$$

where we use Fact 4.2 in the last inequality. Our claim now follows by combining these insights. □

Analyzing distances in H_i. For $s = 0, 1, \ldots, \ell - 2$, by choice of x_{s+1} and y_{s+1}, we have $\widetilde{minPivotDist}_{i+1}(y_s) \le \widetilde{pivotDist}_{i+1}(y_s) \le 4 \cdot dist_{H_i}(y_s, y_{s+1})$ and therefore, using Item 1, also

$$\begin{aligned} \widetilde{minPivotDist}_{i+1}(x_{s+1}) &\le \widetilde{pivotDist}_{i+1}(x_{s+1}) && (1) \\ &\le 4 \cdot dist_{H_i}(x_{s+1}, V(H_{i+1})) && (2) \\ &\le 4 \left(dist_{H_i}(x_{s+1}, y_s) + dist_{H_i}(y_s, p_{i+1}(y_s)) \right) \\ &\le 4 \left(dist_{H_i}(x_{s+1}, y_s) + dist_{H_i}(y_s, \widetilde{pivotDist}_{i+1}(y_s)) \right) \\ &\le 16 \left(dist_{H_i}(x_{s+1}, y_s) + dist_{H_i}(y_s, y_{s+1}) \right) && (3) \\ &= 16 \cdot dist_{H_i}(x_{s+1}, y_{s+1}). \end{aligned}$$

In the last line we used the definition of x_{s+1} and the fact that y_{s+1} is further on π from y_s. In the last equality we have used the fact that π is the shortest path in H_i.

Using again Item 1, we further obtain via the triangle inequality

$$\begin{aligned}\widetilde{minPivotDist}_{i+1}(y_{s+1}) &\leq \widetilde{pivotDist}_{i+1}(y_{s+1}) && (4)\\ &\leq 4 \cdot dist_{H_i}(y_{s+1}, V(H_{i+1})) && (5)\\ &\leq 4 \cdot \left(dist_{H_i}(y_{s+1}, y_s) + dist_{H_i}(y_s, V(H_{i+1}))\right)\\ &\leq 4 \cdot \left(dist_{H_i}(y_{s+1}, y_s) + \widetilde{pivotDist}_{i+1}(y_s)\right) && (6)\\ &\leq 20 \cdot dist_{H_i}(y_{s+1}, y_s).\end{aligned}$$

Analyzing distances in H_{i+1}. Let us recall our analysis: we segmented the path π using the sequence $y_0, x_1, y_1, \ldots, y_{\ell-1}, x_\ell$ into segments in H_i that we then lift to H_{i+1}. The total weight of segments $y_s \rightsquigarrow x_{s+1}$ lifted to H_{i+1} is

$$\sum_{s=0}^{\ell-1} w_{H_{i+1}}(p_{i+1}(y_s), p_{i+1}(x_{s+1})) = 8\sum_{s=0}^{\ell-1} \lceil \widetilde{pivotDist}_{i+1}(y_s)\rceil_2 = 8\sum_{s=1}^{\ell-1} \lceil \widetilde{pivotDist}_{i+1}(y_s)\rceil_2$$

where we use in the last equality that $y_0 = u \in V(H_{i+1})$ which implies that $\widetilde{pivotDist}_{i+1}(y_0) = 0$ by Item 1. It remains to use Equation (5) to deduce that

$$\sum_{s=0}^{\ell-1} w_{H_{i+1}}(p_{i+1}(y_s), p_{i+1}(x_{s+1})) \leq 160 \cdot dist_{H_i}(u, v).$$

For the second type of segments, we can bound the weight of these segments using Lemma 4.4 to obtain

$$\begin{aligned}&\sum_{s=1}^{\ell-1} dist_{H_{i+1}}(p_{i+1}(x_s), p_{i+1}(y_s))\\ &\leq \sum_{s=1}^{\ell-1}\left(69 \cdot w_{H_i}(x_s, y_s) + \sum_{z\in\{x_s,y_s\}} 85 \cdot \lceil \widetilde{minPivotDist}_{i+1}(z)\rceil_2\right)\\ &\leq 69 \cdot dist_{H_i}(u, v) + 85 \cdot (40 \cdot dist_{H_i}(u, v))\\ &\leq 3469 \cdot dist_{H_i}(u, v).\end{aligned}$$

Then by combining the weights of these two segment types we get $dist_{H_{i+1}}(u, v) \leq 3629 \cdot dist_{H_i}(u, v)$, as desired.

4.2 An Algorithm for Maintaining the Hierarchy

The main result of this section is summarized in the following theorem.

Theorem 4.6. *Given an incremental, undirected, weighted graph $G = (V, E, w)$, there is a deterministic algorithm that maintains the hierarchy of vertex sparsifiers $H_1, H_2, \ldots, H_k$ as described in Definition 4.1 for some $k = \Theta(\log\log n)$. Additionally, the algorithm answers queries given a level $1 \leq i \leq k$, and vertices $u, v \in V(H_i)$ where the query returns a distance estimate $\widetilde{dist}(u, v)$ that satisfies $dist_{H_i}(u, v) \leq \widetilde{dist}(u, v)$ and if $u \in B_{H_i}(v, \frac{1}{8}\widetilde{pivotDist}_{i+1}(v))$ or $v \in B_{H_i}(u, \frac{1}{8}\widetilde{pivotDist}_{i+1}(u))$ where we define $\widetilde{pivotDist}_{k+1}(v) = \infty$, it is further guaranteed that $\widetilde{dist}(u, v) \leq 2 \cdot dist_{H_i}(u, v)$.*

The algorithm takes total time $O(kn\log^6 nW + km)$ and answers every query in worst-case $O(1)$ time.

For the rest of this section, we assume w.l.o.g. that G is initially connected, has diameter at most n^2W[8] and all edge weights in $[2, W]$.

Let us now start by giving an algorithm to maintain the approximate pivots $p_{i+1}(v)$ for each $v \in V(H_i)$. This also allows us to determine the vertex sets of each graph H_{i+1}. Once this algorithm is set-up, we give an algorithm to maintain historic approximate pivots $\overline{p}_{i+1}(v)$ for each $v \in V$. Finally, we discuss how to maintain the edges in the graph hierarchy. We note that for technical reasons, all algorithms work on the graphs $\widehat{H}_1, \widehat{H}_2, \ldots, \widehat{H}_k$ where $\widehat{H}_i$ is the graph H_i with all edges rounded up to the nearest power of 2.

Algorithm 2: UpdateApproxPivots()

```
1 for $i = 1, \ldots, k-1$ do
2     while $\exists v \in V(\widehat{H}_i)$ such that
        $|B_{\widehat{H}_i}(v, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))| \geq \hat{b}_i$ do
3         Let $B_v$ be a set of size $\hat{b}_i$ such that
            $B_v \subseteq B_{\widehat{H}_i}(v, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$.
4         if $\exists u \in B_v$ with
            $\widetilde{pivotDist}_{i+1}(u) < \frac{1}{2}\widetilde{pivotDist}_{i+1}(v)$ then
5             $p_{i+1}(v) \leftarrow p_{i+1}(u)$; $\widetilde{pivotDist}_{i+1}(v) \leftarrow$
                $dist_{\widehat{H}_i}(v, u) + \widetilde{pivotDist}_{i+1}(u)$.
6         else
7             Add $v$ to $\widehat{H}_{i+1}$.
8             foreach $u \in B_v$ do
9                 $p_{i+1}(u) \leftarrow v$;
                  $\widetilde{pivotDist}_{i+1}(u) \leftarrow dist_{\widehat{H}_i}(u, v)$.
```

Parameters. Throughout the section, we use parameters $b_i = 2^{(6/5)^i}$ for any $i \geq 1$, and let k be the smallest index such that $\prod_{i\leq k} b_i > n$. It is straight-forward to calculate that $k = \Theta(\log\log n)$. For convenience, we define $\hat{b}_i = b_i \cdot (\log_{4/3}(n^2W) + 1)$ for each i.

Maintaining Approximate Pivots (Pseudo-code). We initialize for each $u \in V$, the pivot $p_2(u) = \perp$ and let $\widetilde{pivotDist}_2(u) = n^2W$. We initialize H_1 to G and $H_2, H_3, \ldots, H_k$ to empty graphs (and initialize $\widehat{H}_1, \widehat{H}_2, \ldots, \widehat{H}_k$ to empty graphs). Throughout the algorithm, whenever a new vertex v is added to vertex set $V(H_i)$ (and thus also to $V(\widehat{H}_i)$), we again initialize its pivot $p_{i+1}(u) = \perp$ and let $\widetilde{pivotDist}_{i+1}(u) = n^2W$.

After this initialization and after each update to G, we invoke UpdateApproxPivots() given in Algorithm 2. The goal of the algorithm is two-fold:

(Ball Size Constraint): Intuitively, the ball $B_{H_i}(v, d_{H_i}(v, V(H_{i+1})))$ should contain at most $\hat{b}_i$ vertices for each $v \in V(H_i)$, so that the

[8]We use n^2W factor, so that we can use the connectivity assumption w.l.o.g. as otherwise we would add a super source with weights nW.

local computation can be done more efficiently. If this constraint is violated, we need to take action and make a new vertex in this ball a pivot of v so that the ball shrinks in size. Since we work with approximate pivots, however, we have to relax this constraint. To counter this, we search even more aggressively for an approximate pivot that enforces this constraint on the ball $B_{\widehat{H}_i}(v, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$. Since $d_{H_i}(v, p_{i+1}(v)) \leq \widetilde{pivotDist}_{i+1}(v) \leq 4 \cdot dist_{H_i}(v, V(H_{i+1}))$ by Item 1 of Definition 4.1, we thus have

$$|B_{\widehat{H}_i}(v, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))| \leq |B_{H_i}(v, d_{H_i}(v, V(H_{i+1})))| \leq \hat{b}_i.$$

(Graph Size Constraint): On the other hand, we also want $V(H_{i+1})$ to be significantly smaller than $V(H_i)$ (roughly by a factor $\hat{b}_i$). Since $V(H_{i+1})$ is in the image of p_{i+1}, we have to make each approximate pivot $p_{i+1}(v)$ a vertex of $V(H_{i+1})$. At an extreme, while making each vertex in $V(H_i)$ its own approximate pivot is a viable choice of p_{i+1} with respect to ball sizes, it is a poor choice when considering the size of $V(H_{i+1})$. Therefore, we use a dynamic covering technique that allows us to bound the number of vertices that are in $V(H_{i+1})$ (i.e. at any point in the image of p_{i+1}) by a much smaller factor.

Our algorithm optimizes these two constraints using a simple rule: whenever a new pivot is required due to a ball size constraint being violated, such a vertex v in need, first asks other vertices that are close to it if their approximate pivot is a good fit. Otherwise, v becomes a pivot itself, but also the new pivot of these close vertices. We note that the algorithm also already shows how to maintain $\widetilde{pivotDist}_{i+1}(v)$ for all $v \in V(H_i)$ for all i. Below, we establish our claim on the graph size.

Claim 4.7. *Whenever the approximate pivot $p_{i+1}(v)$ of a vertex $v \in V(H_i)$ is changed, the value $\widetilde{pivotDist}_{i+1}(v)$ decreases to a $\frac{3}{4}$-fraction of the original value.*

Proof. It is not hard to see from Algorithm 2 that for each $v \in V(H_i)$, $\widetilde{pivotDist}_{i+1}(v)$ is monotonically decreasing over time. Further, when the pivot of a vertex v is changed in Line 5, we have that $\widetilde{pivotDist}_{i+1}^{NEW}(v) = dist_{H_i}(v, u) + \widetilde{pivotDist}_{i+1}(u) \leq \frac{3}{4}\widetilde{pivotDist}_{i+1}^{OLD}(v)$ by the if-condition.

If the pivot of a vertex u is changed in the for-each loop in Line 8, we have that $\widetilde{pivotDist}_{i+1}^{NEW}(u) = dist_{H_i}(u, v) \leq \frac{1}{4}\widetilde{pivotDist}_{i+1}^{OLD}(v) \leq \frac{1}{2}\widetilde{pivotDist}_{i+1}^{OLD}(u)$ where the last inequality stems from Line 3 and the fact that the if condition in Line 4 was not satisfied. □

Corollary 4.8. *At any stage, we have*

$$|V(H_{i+1})| \leq \frac{|V(H_i)|(\log_{4/3}(n^2W) + 1)}{\hat{b}_i} = \frac{|V(H_i)|}{b_i}.$$

Proof. By Claim 4.7, we have that every vertex can change its approximate pivot at most $\log_{4/3}(n^2W) + 1$ times. But note that whenever a new vertex v is added to the set $V(H_{i+1})$ (which happens only in Line 7), we change the approximate pivots of all vertices in the current ball $B_{H_i}(v, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$ which has size at least $\hat{b}_i$ by Line 3. □

For convenience, we define n_i for each level i the final size of set $V(H_i)$ for the rest of this section. Thus, the corollary above can be restated as $n_{i+1} \leq n_i/b_i$.

Maintaining Approximate Pivots (Implementation). Next, we propose an efficient implementation of Algorithm 2 given that there is a procedure that updates $\widehat{H}_i$ based on the updated approximate pivots/ approximate pivot distances.

In our implementation, we use the following crucial primitive TruncDijkstra(v, H_i, r): for any vertex $v \in V(H_i)$ and radius r, we run Dijkstra's algorithm from v in $\widehat{H}_i$ where we stop relaxing vertices that are at distance greater than r, or abort after having found $\hat{b}_i$ such vertices. But note that by definition of the adjacency list Adj$_{\widehat{H}_i,u}$ of each vertex u in graph H_i, we can use exclusively the edges in Adj$_{\widehat{H}_i,u}[1, \hat{b}_i]$ for each u that we relax and therefore implement the algorithm efficiently in $O(\hat{b}_i^2 + \hat{b}_i \log \hat{b}_i)$ time (recall from Section 3 that these lists are ordered by weight and time of arrival). Further, we can store with v a deterministic dictionary $\mathcal{D}_i(v)$ (see [32]) that allows us to check for each vertex u if u is one of the $\hat{b}_i$ vertices relaxed by Dijkstra's algorithm, and if so, we can return the distance $dist_{\widehat{H}_i}(u, v)$. The construction time of the dictionary is subsumed by the bound $O(\hat{b}_i^2 \log \hat{b}_i)$ and its query time is worst-case constant.

Equipped with this primitive, let us give the entire algorithm. Throughout each stage, we maintain a list of *unvisited* vertices Unvisited $\subseteq V(\widehat{H}_i)$ that corresponds to vertices where we cannot currently ensure that the while-loop condition in Line 2 holds.

At the initial stage, we have Unvisited equal to $V(\widehat{H}_i)$, i.e. the initial set of vertices of $\widehat{H}_i$. At the beginning of any subsequent stage, Unvisited consists of the vertices $u \in V(H_i)$ for which there exists a vertex $v \in \mathcal{D}_u$ where Adj$_{\widehat{H}_i,v}[1, \hat{b}_i]$ was updated since the last stage.

Then, at any stage, once Unvisited is initialized as described above, we use the following procedure: while there exists a vertex $v \in$ Unvisited, we run TruncDijkstra$(v, \widehat{H}_i, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$. If the primitive explores less than $\hat{b}_i$ vertices, we store the dictionary $\mathcal{D}_i(v)$ and remove v from Unvisited. Otherwise, i.e. if the primitive explores $\hat{b}_i$ vertices for v, then we enter the while-loop. We can obtain B_v as described in Line 3 from the primitive (by scanning the dictionary) and it is not hard to see that the rest of the while-loop iteration can be implemented in time $O(\hat{b}_i)$.

We prove next that this implementation is efficient.

Lemma 4.9. *Given a procedure that updates each $\widehat{H}_i$ just before the i-th iteration of Algorithm 2 in such a way that for each u keeps the adjacency list of u in $\widehat{H}_i$ ordered by weight and time of arrival, the total update time of all invocations of Algorithm 2 (excluding the time required by the procedure updating each $\widehat{H}_i$) can be bounded by $O\left(\sum_i n_i b_i^4 \log^6 nW + \Delta\right)$ where Δ is the total number of edges and vertices in the final versions of $H_1, H_2, \ldots, H_k$.*

Additionally we can query given a level $1 \leq i \leq k$, and vertices $u, v \in V(H_i)$ where the query returns a distance estimate $\widetilde{dist}(u, v)$ satisfying $dist_{H_i}(u, v) \leq \widetilde{dist}(u, v)$. If $u \in B_{H_i}(v, \frac{1}{8}\widetilde{pivotDist}_{i+1}(v))$ or $v \in B_{H_i}(u, \frac{1}{8}\widetilde{pivotDist}_{i+1}(u))$ where we define $\widetilde{pivotDist}_{k+1}(v) = \infty$, it is further guaranteed that $\widetilde{dist}(u, v) \leq 2 \cdot dist_{H_i}(u, v)$.

Proof. We first argue for correctness. Note that each vertex v that is on the list Unvisited and then removed at the end of the stage cannot satisfy the condition of the while-loop in Line 2. This follows from the fact that a vertex v is only removed from the list UNVISITED when TRUNCDIJKSTRA$(v, \widehat{H}_i, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$ certifies that it cannot satisfy the while-loop condition. Further, we have that $\widetilde{pivotDist}_{i+1}(v)$ is monotonically decreasing over time by Claim 4.7 and therefore the condition remains true. It remains to argue that we can initialize UNVISITED at a stage after the initial stage to only consist of vertices u that had no vertex $v \in \mathcal{D}_u$ with updated ADJ$_{\widehat{H}_i,v}[1, \hat{b}_i]$. But this implies that the primitive TRUNCDIJKSTRA$(u, \widehat{H}_i, \frac{1}{4}\widetilde{pivotDist}_{i+1}(u))$ would relax the same vertices as at the previous stage. It is thus straightforward to prove by induction that for v the while-loop condition does not hold.

We observe first that for each vertex v, its adjacency list ADJ$_{\widehat{H}_i,v}[1, \hat{b}_i]$ can be updated at most $\hat{b}_i \log_2(nW)$ times over the entire course of the algorithm by the way the ordering of edges incident on v is determined and by the fact that $\widehat{H}_i$ only allows for edge weights that are powers of 2 and is incremental.

For the running time, we first use that for each vertex $v \in V(H_i)$, we have that $\widetilde{pivotDist}_{i+1}(v)$ is decreased at most $O(\log(nW))$ times over the entire course of the algorithm by Claim 4.7. Further, between any two times that $\widetilde{pivotDist}_{i+1}(v)$ is decreased, we claim that the primitive TRUNCDIJKSTRA$(v, \widehat{H}_i, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$ is invoked at most $O(\hat{b}_i)$ times. This follows since between these times $\widetilde{pivotDist}_{i+1}(v)$ remains fixed and since $\widehat{H}_i$ is an incremental graph, $B_{\widehat{H}_i}(v, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$ is monotonically increasing. However only until it contains $\hat{b}_i$ or more vertices, as this triggers that the while-loop condition is violated on v again. Until then, each of the at most $\hat{b}_i - 1$ vertices that are in the ball just before it starts violating the while-loop condition can have $\hat{b}_i \log_2(nW)$ updates to their adjacency list ADJ$_{\widehat{H}_i,u}[1, \hat{b}_i]$ triggering an additional invocation of TRUNCDIJKSTRA$(v, \widehat{H}_i, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$. In summary, we bound the number of times TRUNCDIJKSTRA$(v, \widehat{H}_i, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$ is run for some vertex $v \in V(H_i)$ to be at most $O(\hat{b}_i^2 \log^2 nW)$. Using that the time spent by of each such call can be upper bound by $O(\hat{b}_i^2)$, and given that these calls asymptotically subsumes all other operations of the implementation of Algorithm 2, we thus arrive at the runtime stated above.

To prove that we can carry out queries as stated, we require two insights: 1) $dist_{H_i}(u,v) \leq dist_{\widehat{H}_i}(u,v) \leq 2dist_{H_i}(u,v)$ which is trivial from the fact that each $\widehat{H}_i$ is derived from H_i by rounding up weights to the nearest power of 2 and 2) the fact that $V(H_k) = \emptyset$ which follows from Corollary 4.8 which implies $|V(H_k)| \leq n/\prod_{j=1}^{k} b_j$ and the fact that k is chosen such that $\prod_{j=1}^{k} b_j > n$. Given these two insights, it is not hard to verify that the dictionaries $\mathcal{D}_i(v)$ that we have stored at the end of each stage enable us to carry out the stated queries where we obtain for a vertex u the exact distance $dist_{\widehat{H}_i}(u,v)$ and return it as a distance estimate or if we cannot find an entry in the dictionary we can simply return ∞. □

Maintaining Historic Approximate Pivots. Before we can describe how to maintain the graphs $H_1, H_2, \ldots, H_k$ (and thus $\widehat{H}_1, \widehat{H}_2, \ldots, \widehat{H}_k$), it is straight-forward to see from Definition 4.1, that we also need to maintain the last improving pivots $\overline{p}_{i+1}(v)$. Therefore, we need to know the current pivot distances $\widetilde{pivotDist}_{i+1}(v)$ for each $v \in V$ (recall that Algorithm 2 maintains these distances only for vertices in $V(H_i)$). More precisely, we need an algorithm that informs us when $\lceil min\widetilde{PivotDist}_{i+1}(v)\rceil_2$ decreases.

Focusing on a given level $i \leq k-1$, we keep explicit variables $\widetilde{mpd}_{i+1}(v)$ for each $v \in V = V(H_1)$. Our algorithm ensures that we have $\widetilde{mpd}_{i+1}(v) = \lceil min\widetilde{PivotDist}_{i+1}(v)\rceil_2$ by the end of each stage. To achieve this goal, consider the following natural hierarchy forest F_i. We let the vertices of F_i correspond to the vertices in $V(H_1), V(H_2), \ldots, V(H_{i+1})$, where we have an edge from each vertex $v \in V(H_j), j \leq i$ to its current approximate pivot, i.e. an edge $(v, p_{j+1}(v))$. Using directed edges, it is clear that the vertices in $V(H_{i+1})$ form the roots of the forest F_i, the vertices in $V(H_j)$ form the level-j vertices (i.e. at distance $j-1$ from the leaves) and $V(H_1)$ form the leaf vertices.

We further maintain the following weight function w_{F_i} over the edges:

$$w_{F_i}(e = (v, p_{j+1}(v))) = \begin{cases} \widetilde{pivotDist}_{j+1}(v) - \widetilde{mpd}_{i+1}(v) & \text{if j = 1} \\ \widetilde{pivotDist}_{j+1}(v) & \text{otherwise} \end{cases}$$

We can maintain this collection of trees F_i using the dynamic tree data structure introduced below.

THEOREM 4.10 (SEE [6], THEOREM 2.7). *Given a directed forest $F = (V, E, w)$ with (possibly negative) edge weights w, there is a data structure $\mathcal{D}$ that supports the following operations:*

- *ADDEDGE(e, w_e) / DELETEEDGE(e): adds an edge e with weight w_e (assuming that the tail of e is a root) / deletes an edge e from F.*
- *FINDROOT(u): Returns the root of a vertex u.*
- *RETURNDIST(u): Returns the distance from u to its root.*
- *MARK(u) / UNMARK(u): Marks / Unmarks a vertex u. Initially all vertices are unmarked.*
- *FINDNEARESTMARKEDVERTEX(u): Finds the vertex in u's subtree that is at closest distance and marked (if such a vertex exists).*

The data structure can be initialized in $O(n \log n)$ time and implement each operation in $O(\log n)$ time.

We can now state our algorithm to maintain the correct values $\widetilde{mpd}_{i+1}(v)$ for each $v \in V = V(H_1)$. On initialization, we mark all vertices in $V(H_1)$ and leave all other vertices unmarked. We initialize for each $v \in V$, $\widetilde{mpd}_{i+1}(v) = \lceil n^2 W\rceil_2$.

Next, consider an update to G. This potentially results in many changes of edges in F_i and weights due to changes in the distances to approximate pivots $\widetilde{pivotDist}_{j+1}(v)$. Our algorithm starts by forwarding these changes to the data structure $\mathcal{D}$ (for the initial change we assume the data structure is initially empty so the entire forest F_i is encoded in these changes). Weight changes are implemented by first deleting edges and then adding them back into the forest with their new weight.

Then, for each tree $T \in F_i$ that underwent some change at the current stage, we find its root $r \in V(H_{i+1})$ and query for $u =$ FINDNEARESTMARKEDVERTEX(r), the nearest leaf of r. If the distance from u to r is non-negative, the algorithm moves on to the next tree. Otherwise, it sets $\widetilde{mpd}_{i+1}(u) = \lceil min\widetilde{PivotDist}(u)\rceil_2$

which can be extracted from the distance from u to r. Then, the algorithm repeats this the procedure for the current tree.

Claim 4.11. *At the end of each stage, we have for each $v \in V$,*

$$\widetilde{mpd}_{i+1}(v) = \lceil \widetilde{minPivotDist}_{i+1}(v) \rceil_2.$$

Proof. Whenever $\widetilde{mpd}_{i+1}(v)$ is re-set, it is set to the current value $\lceil \widetilde{minPivotDist}(u) \rceil_2$. Since $\lceil \widetilde{minPivotDist}(u) \rceil_2$ is monotonically decreasing over time, we have $\widetilde{mpd}_{i+1}(v) \geq \lceil \widetilde{minPivotDist}(u) \rceil_2$.

On the other hand, by our algorithm, we ensure that at the end of every stage, for each tree $T \in F_i$, that the nearest leaf to its root is at a non-negative distance. Thus, it is not hard to see that for each $v \in V(H_1)$, we have $\sum_{j \leq i} \widetilde{pivotDist}_{j+1}(v) \geq \widetilde{mpd}_{i+1}(v)$. But since $\widetilde{minPivotDist}(u)$ is exactly the minimum over all such sums $\sum_{j \leq i} \widetilde{pivotDist}_{j+1}(v)$ at previous and the current stages, we have that $\widetilde{mpd}_{i+1}(v) \leq \widetilde{minPivotDist}(u)$. Rounding up both quantities to the nearest power of two further preserves this inequality. ◻

Claim 4.12. *Given an algorithm Algorithm 2 to update for each $1 \leq i < k$, $v \in V(H_i)$ the quantities $\widetilde{pivotDist}_{i+1}(v)$ and $p_{i+1}(v)$, we can maintain for each $1 \leq i < k$ and $v \in V$, the last improving pivots $\overline{p}_{i+1}(v)$ and $\lceil \widetilde{minPivotDist}_{i+1}(v) \rceil_2$ in additional time $O(kn \log nW \log n)$.*

Proof. Fixing a level $1 \leq i < k$, we have that each vertex in $V(H_i)$ has $\widetilde{pivotDist}_{i+1}(v)$ and $p_{i+1}(v)$ updated at most $O(\log nW)$ times. Since each update can be handled by the algorithm described above in time $O(\log n)$, the algorithm takes $O(|V(H_i)| \log nW \log n)$ time on handling updates and the resulting query on the updated tree. Further, each time a query returns a negative value, we at least half the value $\widetilde{mpd}_{i+1}(u)$ for some $u \in V$. Thus, the number of such queries is bound by $O(n \log nW)$ and each query and subsequent update of $\widetilde{mpd}_{i+1}(u)$ can again be implemented in time $O(\log n)$. Since we have $k - 1$ levels where we maintain our data structure, the bound follows. ◻

Putting it all together. Finally, we give the algorithm to maintain $H_1, H_2, \ldots, H_k$ (and thus to maintain $\widehat{H}_1, \widehat{H}_2, \ldots, \widehat{H}_k$). Recall that initially these graphs are equal to the empty graphs. Then, H_i (and $\widehat{H}_i$) is updated whenever Algorithm 2 is invoked, and to be precise, is updated just before the i-th iteration of the for-loop in Algorithm 2. For $i = 1$, the update is simple as H_1 is just equal to G at any stage. For $i + 1 \geq 1$, we have that H_i is already updated for the current stage, and for any $v \in V(H_i)$, we have $p_{i+1}(v), \widetilde{pivotDist}_{i+1}(v), B_{H_i}(v, \frac{1}{4}\widetilde{pivotDist}_{i+1}(v))$ and $\mathcal{D}_v$, and for any $v \in V$, we have $\overline{p}_{i+1}(v)$ and $\lceil \widetilde{minPivotDist}_{i+1}(v) \rceil_2$ in their updated version (i.e. these values do not change for the rest of the stage). This can be seen easily from Algorithm 2 and our description of the algorithm to maintain historic approximate pivots.

Given this updated information, it is straight-forward to generate all edges that are missing from H_{i+1} in constant additional time per edge added to H_{i+1}. To obtain a bound on the runtime, it thus suffices to bound the number of edges in H_{i+1}.

Lemma 4.13. *Throughout the algorithm, we have for any $1 \leq i < k$, that $|E(H_{i+1})| = O(m + kn \log^5 nW + \sum_{j \leq i} |V(H_j)| b_j \log^4 nW)$.*

Proof. We first prove the claim that for any such i, we have $|E_{i+1}^{base}| = O(n \log^3 nW + |V(H_i)| \hat{b}_i \log nW)$. We proceed by a case analysis for each edge type that is generated. Following Definition 4.1 we have

- For edges generated from Item 3, we have that there is at most one edge generated for each vertex v in $B_{H_i}(u, \frac{1}{4} \cdot \widetilde{pivotDist}_{i+1}(u))$ for any vertex $u \in V(H_i)$. Since the quantity $\widetilde{pivotDist}_{i+1}(u)$ is updated at most $O(\log nW)$ times and since it is ensured that $B_{H_i}(u, \frac{1}{4} \cdot \widetilde{pivotDist}_{i+1}(u))$ is of size less than $\hat{b}_i$ when edges are generated, there are at most $O(|V(H_i)| \hat{b}_i \log nW)$ such edges.
- For edges generated from Item 4, we have that such edges are only generated for a vertex $v \in V(H_i)$ whenever its pivot $p_{i+1}(v)$ is updated, and if so an edge to every former pivot of v is added. But since we have from Claim 4.7 that there are at most $O(\log nW)$ such pivots throughout the algorithm, the total number of such edges can be bound by $O(|V(H_i)| \log^2 nW)$.
- For edges generated from Item 5, we have to generate new edges for a vertex $v \in V$ only when $\overline{p}_{i+1}(v)$ is updated or if for an existing edge $(\overline{p}_{i+1}^{(t)}(x), \overline{p}_{i+1}^{(t)}(v))$ where $x = \overline{p}_i^{(t')}(v)$ for some $t' \leq t$, the weight has to change because of one of the quantities $\lceil \widetilde{minPivotDist}_{i+1}^{(t)}(x) \rceil_2$, $\lceil \widetilde{minPivotDist}_{i}^{(t')}(v) \rceil_2$, or $\lceil \widetilde{minPivotDist}_{i+1}^{(t)}(v) \rceil_2$. By our previous argument, we have that there are at most $O(\log nW)$ historic pivots $\overline{p}_i(v), \overline{p}_{i+1}(v)$ and that $\lceil \widetilde{minPivotDist}_{i+1}^{(t)}(x) \rceil_2$, $\lceil \widetilde{minPivotDist}_{i+1}^{(t)}(v) \rceil_2$ can change at most $O(\log nW)$ times for each pair x, v for which we generate an edge. We conclude that there are at most $O(n \log^3 nW)$ such edges.

Having established the upper bound on E_j^{base} for all $1 \leq j < k$, and using that E_1^{base} consists of the edges in G, we can thus bound the number of edges in E_{i+1}^{proj}. These edges are generated from Item 6, where we have an edge $(\overline{p}_{i+1}(x), \overline{p}_{i+1}(y))$ in E_{i+1}^{proj} for each edge $(x, y) \in E_j^{base}$ for any $j \leq i$ for any historic pivots of x and y and the edge weight is equal to $\lceil \widetilde{minPivotDist}_{i+1}(x) \rceil_2 + \lceil w_{H_j}(e) \rceil_2 + \lceil \widetilde{minPivotDist}_{i+1}(y) \rceil_2$. From our previous discussion, we have that for each such edge in E_j^{base} for any $j \leq i$, we can have at most $O(\log^2 nW)$ versions in E_{i+1}^{proj}. Thus, the total number of such edges is $O(m + kn \log^5 nW + \sum_{j \leq i} |V(H_j)| \hat{b}_j \log^3 nW)$, as desired. ◻

Using this upper bound on the number of edges, we can prove the main result of the section, Theorem 4.6.

Theorem 4.6. *Given an incremental, undirected, weighted graph $G = (V, E, w)$, there is a deterministic algorithm that maintains the hierarchy of vertex sparsifiers $H_1, H_2, \ldots, H_k$ as described in Definition 4.1 for some $k = \Theta(\log \log n)$. Additionally, the algorithm answers queries given a level $1 \leq i \leq k$, and vertices $u, v \in V(H_i)$ where the query returns a distance estimate $\widetilde{dist}(u, v)$ that satisfies $dist_{H_i}(u, v) \leq \widetilde{dist}(u, v)$ and if $u \in B_{H_i}(v, \frac{1}{8}\widetilde{pivotDist}_{i+1}(v))$ or $v \in B_{H_i}(u, \frac{1}{8}\widetilde{pivotDist}_{i+1}(u))$ where we define $\widetilde{pivotDist}_{k+1}(v) = \infty$, it is further guaranteed that $\widetilde{dist}(u, v) \leq 2 \cdot dist_{H_i}(u, v)$.*

The algorithm takes total time $O(kn\log^6 nW + km)$ and answers every query in worst-case $O(1)$ time.

Proof. We have correctness of the algorithm by Lemma 4.9, Claim 4.11 and our previous discussion. It thus remains only to bound the runtime. From Lemma 4.9, Claim 4.12 and Lemma 4.13, we can upper bound the total update time of the algorithm by

$$O\left(\sum_{i=1}^{k} n_i b_i^4 \log^6 nW + mk\log(m)\right)$$

(recall $n_1 = \Omega(n)$). Further note that for $i < 10$, we have $b_i \le 2^{(6/5)^{10}} = O(1)$ and therefore $n_i b_i^4 = O(n)$. For $i \ge 10$, we have that

$$n_i \le n/\prod_{j=1}^{i-1} b_j = n/2^{\sum_{j=1}^{i-1}(6/5)^j} = n/2^{(6/5)^i \sum_{j=1}^{i-1}(5/6)^j} \le n/b_i^4$$

where we use the formula for geometric sums to obtain $\sum_{j=1}^{i-1}(5/6)^j \ge (1-(5/6)^{-10})/(1-5/6) - 1 \ge 4$. This allows to bound the total update time as stated above, and it remains to observe that one can implement the data structure to maintain Adj_{v,H_i} for each i in time $O(m\log m)$ by using binary search when adding new edges in to the adjacency list of v. □

4.3 The Query Algorithm

Finally, we can give a query algorithm that is almost identical to the query algorithm in [7]. Here we define for convenience the function $p_1 : V \mapsto V$ to be the identity function and recall that $\widetilde{pivotDist}_{k+1}(x) = \infty$ for all $x \in V(H_k)$.

Algorithm 3: QueryDist(u, v)

```
1 $i \leftarrow 1$.
2 while the distance estimate $\widetilde{dist}(p_i(u), p_i(v))$ from Theorem 4.6 exceeds
  $\max\{\frac{1}{4}\widetilde{pivotDist}_{i+1}(p_i(v)), \frac{1}{4}\widetilde{pivotDist}_{i+1}(p_i(u))\}$ do
3     $\tilde{d}_i \leftarrow \widetilde{pivotDist}_{i+1}(p_i(u)) + \widetilde{pivotDist}_{i+1}(p_i(v))$.
4     $i \leftarrow i + 1$.
5 $\tilde{d}_i \leftarrow \widetilde{dist}(p_i(u), p_i(v))$.
6 return $\widetilde{dist}(u, v) = \sum_{j\le i} \tilde{d}_j$
```

Lemma 4.14. *The algorithm QueryDist(u, v) returns a distance estimate $\widetilde{dist}(u, v)$ such that*

$$dist_G(u, v) \le \widetilde{dist}(u, v) \le \tilde{O}(1)\cdot dist_G(u, v).$$

The algorithm runs in worst-case time $O(\log\log n)$.

Runtime Analysis. To bound the runtime, we first observe that if we are in the j-th iteration of the while-loop, once we evaluated $p_j(u)$ and $p_j(v)$, we can implement the while-loop in $O(1)$ time. While $p_j(u)$ and $p_j(v)$ are nested functions of depth $j-1$ which would naively take time $O(j)$ to evaluate, we use that in the $j-1$-th iteration (for $j > 1$, otherwise $O(j)$ is constant), we already evaluate $p_{j-1}(u)$ and $p_{j-1}(v)$, and by keeping them stored in a cache, we can compute $p_j(u) = p_j(p_{j-1}(u))$ and $p_j(v) = p_j(p_{j-1}(v))$ in constant time respectively. Thus, each iteration of the while loop takes constant time.

Letting i refer to the final value of the variable, we claim that $i \le k$. Given this claim, it is not hard to see that from Algorithm 3 and Theorem 4.6, that the total time spend can be bound by $O(k) = O(\log\log n)$.

To prove that $i \le k$, we observe that the variable is initialized to 1 and then increased by each iteration of the while-loop by just one. Thus, if we assume for the sake of contradiction that $i > k$, the condition of the while-loop before the k-th iteration must have been true. In particular, we have $\widetilde{dist}(p_k(u), p_k(v))$ exceeding $\frac{1}{4}\widetilde{pivotDist}_{k+1}(p_k(v))$ but this gives an immediate contradiction as we define $\widetilde{pivotDist}_{k+1}(x) = \infty$ for all $x \in V(H_i)$ while we assume that H_k is a connected graph.

Lower Bounding the Estimate. Observe that

$$\begin{aligned}\widetilde{dist}(u,v) = \sum_{j\le i}\tilde{d}_j &\ge \sum_{\ell=1}^{i-1}\widetilde{pivotDist}_{\ell+1}(p_\ell(u)) + dist_{H_i}(p_i(u), p_i(v))\\ &+ \sum_{\ell=1}^{i-1}\widetilde{pivotDist}_{\ell+1}(p_\ell(v))\\ &\ge \sum_{\ell=1}^{i-1} dist_{H_\ell}(p_\ell(u), p_{\ell+1}(u)) + dist_{H_i}(p_i(u), p_i(v))\\ &+ \sum_{\ell=1}^{i-1} dist_{H_\ell}(p_\ell(v), p_{\ell+1}(v))\\ &\ge \sum_{\ell=1}^{i-1} dist_G(p_\ell(u), p_{\ell+1}(u)) + dist_G(p_i(u), p_i(v))\\ &+ \sum_{\ell=1}^{i-1} dist_G(p_\ell(v), p_{\ell+1}(v)) \ge dist_G(u,v)\end{aligned}$$

where we have the first equality from Line 3 and Line 5, the second inequality follows from Item 1, the third inequality from Theorem 4.3, and the final inequality from the triangle inequality.

Upper Bounding the Estimate. We next prove by induction that for each $1 \le \ell \le i$, we have $\sum_{j=\ell}^{i}\tilde{d}_j \le 2\cdot 235885^{i-\ell} dist_{H_i}(p_i(u), p_i(v))$. For the base case, we have $\ell = i$, and $\tilde{d}_\ell \le 2\cdot dist_{H_\ell}(p_\ell(u), p_\ell(v))$ by Theorem 4.6 which exactly corresponds to the definition in Line 5.

We can then take the inductive step for $\ell + 1 \mapsto \ell$ for some $\ell < i$. From the while-loop condition in Line 2, Theorem 4.6 and the fact that $\ell < i$,

$$\begin{aligned}&dist_{H_\ell}(p_\ell(u), p_\ell(v))\\ &\quad\ge \frac{1}{8}\max\{\widetilde{pivotDist}_{\ell+1}(p_\ell(u)), \widetilde{pivotDist}_{\ell+1}(p_\ell(v))\}\\ &\quad\ge \frac{1}{8}\max\{dist_{H_\ell}(p_\ell(u), p_{\ell+1}(u)), dist_{H_\ell}(p_\ell(v), p_{\ell+1}(v))\}\end{aligned}$$

where the last inequality follows from Item 1. We further obtain

$$\begin{aligned}\sum_{j=\ell}^{i} \widetilde{d}_j &= \sum_{j=\ell+1}^{i} \widetilde{d}_j + \widetilde{d}_\ell \\ &\leq 2 \cdot 235885^{i-\ell-1} dist_{H_{\ell+1}}(p_{\ell+1}(u), p_{\ell+1}(v)) + \widetilde{d}_\ell \\ &\leq 2 \cdot 3629 \cdot 235885^{i-\ell-1} dist_{H_\ell}(p_{\ell+1}(u), p_{\ell+1}(v)) + \widetilde{d}_\ell \\ &\leq 2 \cdot 3629 \cdot 236145^{i-\ell-1} (dist_{H_\ell}(p_\ell(u), p_\ell(v)) \\ &+ \sum_{x \in \{u,v\}} dist_{H_\ell}(p_{\ell+1}(x), p_\ell(x))) + \widetilde{d}_\ell \\ &\leq 2 \cdot 3629 \cdot 235885^{i-\ell-1} (dist_{H_\ell}(p_\ell(u), p_\ell(v)) \\ &+ \sum_{x \in \{u,v\}} dist_{H_\ell}(p_{\ell+1}(x), p_\ell(x))) \\ &\leq 2 \cdot 3629 \cdot 235885^{i-\ell-1} \cdot 65 \cdot dist_{H_\ell}(p_\ell(u), p_\ell(v)) \\ &= 2 \cdot 235885^{i-\ell} \cdot dist_{H_\ell}(p_\ell(u), p_\ell(v))\end{aligned}$$

where we use the induction hypothesis in the first inequality, Theorem 4.3 in the second inequality, the triangle inequality in the third inequality. In the fourth inequality, we use that by Line 3 and Item 1 of Definition 4.1, $\tilde{d}_\ell = \widetilde{pivotDist}_{\ell+1}(p_\ell(u)) + \widetilde{pivotDist}_{\ell+1}(p_\ell(v)) \leq 4\left(dist_{H_\ell}(p_\ell(u), p_{\ell+1}(u)) + dist_{H_\ell}(p_\ell(v), p_{\ell+1}(v))\right)$ and the previous statement. Finally, we use our insight from before. This concludes the induction.

As we have established the claim, and bound $i \leq k$, it suffices to see that this implies in particular that $\widetilde{dist}(u,v) = \sum_{j=1}^{i} \widetilde{d}_j \leq 2 \cdot 235885^{i-1} dist_{H_1}(p_1(u), p_1(v)) = 2 \cdot 235885^{i-1} dist_G(u,v) \leq 2 \cdot 235885^{O(\log\log n)} dist_G(u,v) \leq \tilde{O}(1) \cdot dist_G(u,v)$.

5 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers of STOC for their invaluable feedback. Sebastian Forster and Yasamin Nazari are supported by the Austrian Science Fund (FWF): P 32863-N. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 947702). Maximilian Probst Gutenberg has received funding from the grant "Algorithms and complexity for high-accuracy flows and convex optimization" (no. 200021 204787) of the Swiss National Science Foundation.

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. In *Proc. of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2022)*. 1487–1500. https://doi.org/10.1145/3519935.3520066 arXiv:2204.10465

[2] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *Proc. of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2014)*. 434–443. https://doi.org/10.1109/FOCS.2014.53 arXiv:1402.0054

[3] Ittai Abraham and Shiri Chechik. 2013. Dynamic Decremental Approximate Distance Oracles with $(1+\epsilon, 2)$ stretch. *CoRR* abs/1307.1516 (2013). arXiv:1307.1516

[4] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. 2017. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proc. of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*. 440–452. https://doi.org/10.1137/1.9781611974782.28 arXiv:1607.05132

[5] Ittai Abraham, Shiri Chechik, and Kunal Talwar. 2014. Fully Dynamic All-Pairs Shortest Paths: Breaking the $O(n)$ Barrier. In *Proc. of the 17th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX 2014)*. 1–16. https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2014.1

[6] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2005. Maintaining information in fully dynamic trees with top trees. *Acm Transactions on Algorithms* 1, 2 (2005), 243–264.

[7] Alexandr Andoni, Clifford Stein, and Peilin Zhong. 2020. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 322–335.

[8] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. 2007. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *Journal of Algorithms* 62, 2 (2007), 74–92. https://doi.org/10.1016/j.jalgor.2004.08.004 Announced at STOC 2002.

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

[10] Shai Ben-David, Allan Borodin, Richard M. Karp, Gábor Tardos, and Avi Wigderson. 1994. On the Power of Randomization in On-Line Algorithms. *Algorithmica* 11, 1 (1994), 2–14. https://doi.org/10.1007/BF01294260 Announced at STOC 1990.

[11] Thiago Bergamaschi, Monika Henzinger, Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. 2021. New techniques and fine-grained hardness for dynamic near-additive spanners. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 1836–1855.

[12] Aaron Bernstein. 2009. Fully Dynamic $(2+\epsilon)$ Approximate All-Pairs Shortest Paths with Fast Query and Close to Linear Update Time. In *Proc. of the 50th Annual IEEE Symposium on Foundations of Computer Science, (FOCS 2009)*. 693–702. https://doi.org/10.1109/FOCS.2009.16

[13] Aaron Bernstein. 2016. Maintaining Shortest Paths Under Deletions in Weighted Directed Graphs. *SIAM J. Comput.* 45, 2 (2016), 548–574. https://doi.org/10.1137/130938670 Announced at STOC 2013.

[14] Aaron Bernstein and Shiri Chechik. 2016. Deterministic decremental single source shortest paths: beyond the $O(mn)$ bound. In *Proc. of the 48th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2016)*. 389–397. https://doi.org/10.1145/2897518.2897521

[15] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. 2021. Deterministic Decremental SSSP and Approximate Min-Cost Flow in Almost-Linear Time. In *Proc. of the 62nd IEEE Annual Symposium on Foundations of Computer Science, (FOCS 2021)*. 1000–1008. https://doi.org/10.1109/FOCS52979.2021.00100 arXiv:2101.07149

[16] Aaron Bernstein and Liam Roditty. 2011. Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions. In *Proc. of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2011)*. 1355–1365. https://doi.org/10.1137/1.9781611973082.104

[17] Shiri Chechik. 2018. Near-Optimal Approximate Decremental All Pairs Shortest Paths. In *Proc. of the 59th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2018)*. 170–181. https://doi.org/10.1109/FOCS.2018.00025

[18] Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. 2020. Fast Dynamic Cuts, Distances and Effective Resistances via Vertex Sparsifiers. In *Proc. of the 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS 2020)*. 1135–1146. https://doi.org/10.1109/FOCS46700.2020.00109 arXiv:2005.02368

[19] Li Chen, Rasmus Kyng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2023. A Simple Framework for Finding Balanced Sparse Cuts via APSP. In *2023 Symposium on Simplicity in Algorithms, SOSA 2023, Florence, Italy, January 23-25, 2023*, Telikepalli Kavitha and Kurt Mehlhorn (Eds.). SIAM, 42–55. https://doi.org/10.1137/1.9781611977585.ch5

[20] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2022. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*. IEEE, 612–623. https://doi.org/10.1109/FOCS54457.2022.00064

[21] Julia Chuzhoy. 2021. Decremental all-pairs shortest paths in deterministic near-linear time. In *Proc. of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2021)*. 626–639. https://doi.org/10.1145/3406325.3451025 arXiv:2109.05621

[22] 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 *Proc. of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC 2019)*. ACM, 389–400. https://doi.org/10.1145/3313276.3316320 arXiv:1905.11512

[23] Julia Chuzhoy and Thatchaphol Saranurak. 2021. Deterministic Algorithms for Decremental Shortest Paths via Layered Core Decomposition. In *Proc. of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*. 2478–2496. https://doi.org/10.1137/1.9781611976465.147 arXiv:2009.08479

[24] Camil Demetrescu and Giuseppe F. Italiano. 2004. A new approach to dynamic all pairs shortest paths. *Journal of the ACM* 51, 6 (2004), 968–992. https://doi.org/10.1145/1039488.1039492 Announced at STOC 2003.

[25] Camil Demetrescu and Giuseppe F. Italiano. 2006. Fully dynamic all pairs shortest paths with real edge weights. *J. Comput. System Sci.* 72, 5 (2006), 813–837. https://doi.org/10.1016/j.jcss.2005.05.005 Announced at FOCS 2001.

[26] Michal Dory, Sebastian Forster, Yasamin Nazari, and Tijn de Vos. 2022. New Tradeoffs for Decremental Approximate All-Pairs Shortest Paths. *CoRR* abs/2211.01152 (2022). https://doi.org/10.48550/arXiv.2211.01152 arXiv:2211.01152

[27] Paul Erdős. 1963. Extremal problems in graph theory. In *Proc. of the Symposium on Theory of Graphs and its Applications)*. 2936.

[28] Jacob Evald, Viktor Fredslund-Hansen, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. 2021. Decremental APSP in Unweighted Digraphs Versus an Adaptive Adversary. In *Proc. of the 48th International Colloquium on Automata, Languages, and Programming, (ICALP 2021)*. 64:1–64:20. https://doi.org/10.4230/LIPIcs.ICALP.2021.64 arXiv:2010.00937

[29] Shimon Even and Yossi Shiloach. 1981. An On-Line Edge-Deletion Problem. *J. ACM* 28, 1 (1981), 1–4. https://doi.org/10.1145/322234.322235

[30] Sebastian Forster, Gramoz Goranci, and Monika Henzinger. 2021. Dynamic Maintenance of Low-Stretch Probabilistic Tree Embeddings with Applications. In *Proc. of the 2021 ACM-SIAM Symposium on Discrete Algorithms, (SODA 2021)*. 1226–1245. https://doi.org/10.1137/1.9781611976465.75 arXiv:2004.10319

[31] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. 2020. Fully-Dynamic All-Pairs Shortest Paths: Improved Worst-Case Time and Space Bounds. In *Proc. of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*. 2562–2574. https://doi.org/10.1137/1.9781611975994.156 arXiv:2001.10801

[32] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. 2001. Deterministic Dictionaries. *Journal of Algorithms* 41, 1 (2001), 69–85. https://doi.org/10.1006/jagm.2001.1171

[33] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2014. A Subquadratic-Time Algorithm for Decremental Single-Source Shortest Paths. In *Proc. of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*. 1053–1072. https://doi.org/10.1137/1.9781611973402.79

[34] 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. https://doi.org/10.1137/140957299 arXiv:1308.0776 Announced at FOCS 2013.

[35] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2018. Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time. *Journal of the ACM* 65, 6 (2018), 36:1–36:40. https://doi.org/10.1145/3218657 Announced at FOCS 2014.

[36] 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 *Proc. of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC 2015)*. 21–30. https://doi.org/10.1145/2746539.2746609 arXiv:1511.06773

[37] Adam Karczmarz and Jakub Łącki. 2019. Reliable Hubs for Partially-Dynamic All-Pairs Shortest Paths in Directed Graphs. In *Proc. of the 27th Annual European Symposium on Algorithms (ESA 2019)*. 65:1–65:15. https://doi.org/10.4230/LIPIcs.ESA.2019.65 arXiv:1907.02266

[38] Adam Karczmarz, Anish Mukherjee, and Piotr Sankowski. 2022. Subquadratic Dynamic Path Reporting in Directed Graphs Against an Adaptive Adversary. *STOC 2022* (2022).

[39] Valerie King. 1999. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*. 81–91. https://doi.org/10.1109/SFFCS.1999.814580

[40] Jakub Łącki and Yasamin Nazari. 2022. Near-Optimal Decremental Hopsets with Applications. In *Proc. of the 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, Vol. 229. 86:1–86:20. https://doi.org/10.4230/LIPIcs.ICALP.2022.86 arXiv:2009.08416

[41] Aleksander Madry. 2010. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proc. of the 42nd ACM Symposium on Theory of Computing, (STOC 2010)*. 121–130. https://doi.org/10.1145/1806689.1806708 arXiv:1003.5907

[42] Mihai Patrascu. 2010. Towards polynomial lower bounds for dynamic problems. In *Proc. of the 42nd ACM Symposium on Theory of Computing (STOC 2010)*. 603–610. https://doi.org/10.1145/1806689.1806772

[43] 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 Announced at ESA 2004.

[44] Liam Roditty and Uri Zwick. 2012. Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs. *SIAM J. Comput.* 41, 3 (2012), 670–683. https://doi.org/10.1137/090776573 Announced at FOCS 2004.

[45] Piotr Sankowski. 2005. Subquadratic Algorithm for Dynamic Shortest Distances. In *Proc. of the 11th Annual International Computing and Combinatorics Conference (COCOON)*. 461–470. https://doi.org/10.1007/11533719_47

[46] Mikkel Thorup. 2004. Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles. In *Proc. of the 9th Scandinavian Workshop on Algorithm Theory (SWAT 2004)*. 384–396. https://doi.org/10.1007/978-3-540-27810-8_33

[47] Mikkel Thorup. 2005. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proc. of the 37th Annual ACM Symposium on Theory of Computing (STOC 2005)*. 112–119. https://doi.org/10.1145/1060590.1060607

[48] Mikkel Thorup and Uri Zwick. 2005. Approximate distance oracles. *Journal of the ACM* 52, 1 (2005), 1–24. https://doi.org/10.1145/1044731.1044732 Announced at STOC 2021.

[49] Jan van den Brand, Sebastian Forster, and Yasamin Nazari. 2022. Fast Deterministic Fully Dynamic Distance Approximation. In *Proc. of the 63rd IEEE Annual Symposium on Foundations of Computer Science, (FOCS 2022)*. arXiv:2111.03361

[50] Jan van den Brand and Danupon Nanongkai. 2019. Dynamic Approximate Shortest Paths and Beyond: Subquadratic and Worst-Case Update Time. In *Proc. of the 60th IEEE Annual Symposium on Foundations of Computer Science, (FOCS 2019)*. 436–455. https://doi.org/10.1109/FOCS.2019.00035 arXiv:1909.10850

[51] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. 2019. Dynamic Matrix Inverse: Improved Algorithms and Matching Conditional Lower Bounds. In *Proc. of the 60th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2019)*. 456–480. https://doi.org/10.1109/FOCS.2019.00036 arXiv:1905.05067

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