Fast Deterministic Fully Dynamic Distance Approximation

Jan van den Brand
Georgia Institute of Technology
United States
vdbrand@gatech.edu

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

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

***Abstract*—In this paper, we develop deterministic fully dynamic algorithms for computing approximate distances in a graph with worst-case update time guarantees. In particular, we obtain improved dynamic algorithms that, given an unweighted and undirected graph $G = (V, E)$ undergoing edge insertions and deletions, and a parameter $0 < \epsilon \leq 1$, maintain $(1 + \epsilon)$-approximations of the st-distance between a given pair of nodes s and t, the distances from a single source to all nodes ("SSSP"), the distances from multiple sources to all nodes ("MSSP"), or the distances between all nodes ("APSP").**

Our main result is a deterministic algorithm for maintaining $(1 + \epsilon)$-approximate st-distance with worst-case update time $O(n^{1.407})$ (for the current best known bound on the matrix multiplication exponent ω). This even improves upon the fastest known randomized algorithm for this problem. Similar to several other well-studied dynamic problems whose state-of-the-art worst-case update time is $O(n^{1.407})$, this matches a conditional lower bound [BNS, FOCS 2019]. We further give a deterministic algorithm for maintaining $(1 + \epsilon)$-approximate single-source distances with worst-case update time $O(n^{1.529})$, which also matches a conditional lower bound.

At the core, our approach is to combine algebraic distance maintenance data structures with near-additive emulator constructions. This also leads to novel dynamic algorithms for maintaining $(1 + \epsilon, \beta)$-emulators that improve upon the state of the art, which might be of independent interest. Our techniques also lead to improved randomized algorithms for several problems such as exact st-distances and diameter approximation.

***Index Terms*—Graph Algorithms, Data Structures**

I. Introduction

From the procedural point of view, an algorithm is a set of instructions that outputs the result of a computational task for a given input. This static viewpoint neglects that computation is often not a one-time task with input data in successive runs of the algorithm being very similar. The idea of dynamic graph algorithms is to explicitly model the situation that the input is constantly undergoing changes and the algorithm needs to adapt its output after each change to the input. This paradigm has been highly successfully applied to the domain of graph algorithms. The major goal in designing dynamic graph algorithms is to spend as little computation time as possible for processing each update to the input graph.

Despite the progress on dynamic graph algorithms in recent years, many state-of-the-art solutions suffer from at least one of the following restrictions: (1) Many dynamic algorithms only support one type of updates, i.e., are incremental (supporting only insertions) or decremental (supporting only deletions). *Fully dynamic* algorithms support both types of updates. (2) Many dynamic algorithms only achieve amortized update time guarantees, i.e., the stated bound only holds "on average" over a sequence of updates with individual updates possibly taking significantly more time than the stated amortized bound. Worst-case bounds also hold for individual updates, which for example is relevant in real-time systems. (3) Many dynamic algorithms are randomized. (i) On one hand, this means these algorithms only give probabilistic guarantees on correctness or running time that do not hold in all cases. (ii) On the other hand, randomized algorithms often do not allow the "adversary" creating the sequence of updates to be *adaptive* in the sense that it may react to the outputs of the algorithm[1]. This is because the power of randomization can in many cases only be unleashed if the adversary is *oblivious* to the outputs of the algorithm, which guarantees probabilistic independence of the random choices made by the algorithm. *Deterministic* algorithms avoid these two issues.

While these restrictions are not prohibitive in certain settings, they obstruct the general-purpose usage of dynamic algorithms as "black boxes". Thus, the "gold standard" in the design of dynamic algorithms should be deterministic fully dynamic algorithms with worst-case update time bounds. To date, there is only a limited number of problems that admit such algorithms and additionally have time bounds that match (conditional) lower bounds (say up to subpolynomial factors). To the best of our knowledge, this is the case only for $(2 + \epsilon)$-approximate maximum fractional matching and minimum vertex cover [3], $(2\Delta-1)$-edge coloring [4], $(1-\epsilon)$-approximate densest subgraph [5], connectivity [6], minimum

Th full version is available as [1]. This research was done while Jan van den Brand was at the Simons Institute for the Theory of Computing and funded by ONR BRC grant N00014-18-1-2562 and by the Simons Institute through a Simons-Berkeley Postdoctoral Fellowship. 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).

[1]This type of adversary is called "adaptive online adversary" in the context of online algorithms [2]. Note that despite being allowed to choose the next update in its sequence based on the outputs of the algorithm so far, this adversary may not explicitly observe the internal random choices of the algorithm.

2575-8454/22/$31.00 ©2022 IEEE
DOI 10.1109/FOCS54457.2022.00099

spanning tree [6], and edge connectivity [7].

In this paper, we add two important problems to this list: $(1+\epsilon)$-approximate st distances and $(1+\epsilon)$-approximate single-source distances in unweighted, undirected graphs. For current bounds on the matrix-multiplication exponent[2] ω, our deterministic worst-case update times for these problems are $O(n^{1.407})$ and $O(n^{1.529})$, respectively, and match conditional lower bounds from [12] up to subpolynomial factors. In particular, the dynamic $(1+\epsilon)$-approximate st distance currently shares this conditional lower bound and the upper bound we derive with an array of other dynamic problems such as st reachability and cycle detection in directed graphs, maximum matching size, or determinant and rank of a matrix [12]–[15].

Apart from our main results for st and single-source distances, we also obtain novel results for approximating multi-source distances, all-pairs distances, and the diameter; see Section I-A for a detailed overview on our results.

Summarized in one sentence, our results are obtained by combining algebraic bounded-distance data structures with near-additive emulator constructions (see Definition 1.3) and then obtaining distance estimates from such an emulator. A similar strategy was employed by recent related work of [16]. One major ingredient of their approach is equipping the algebraic distance data structure of Sankowski [13], [14] with a path-reporting mechanism similar to Seidel's technique for APSP in the static setting [17]. This allows them to maintain a near-additive spanner – which fits their path-reporting purposes – but together with other parts of their algorithm introduces randomization. By using certain types of emulators instead of spanners, we can obtain a faster, deterministic algorithm. In particular, we can tailor the algebraic data structures better to our needs due to several nice properties of our emulators, for example that their structure changes *slowly* and *locally*.

In the remainder of this section we state all our results and compare them with related work. In Section II, we give an overview of our main ideas and technical contributions. In the full version we provide all the omitted technical details.

A. Our Results and Comparison with Related Work

In this section, we summarize our main results for deterministic fully dynamic distance computation (st, SSSP, APSP, and MSSP supporting distance queries) and emulators. A summary of our algorithms for maintaining $(1+\epsilon)$-approximate distances with their worst-case update time guarantees can be found in Table I. In addition to these deterministic results, our techniques also give improved randomized solutions for diameter approximation, and subquadratic update-time $(1+\epsilon)$-APSP distance oracles[3] with sublinear query time. We next discuss each of these results and compare them with related work. Throughout this paper we assume that we are given an unweighted graph with n nodes and m edges.

Approx	Type	Worst-case update	Reference
$1+\epsilon$	st	$O(n^{1.407}\epsilon^{-2}\log\epsilon^{-1})$	Theorem 1.1
$1+\epsilon$	SSSP	$O(n^{1.529}\epsilon^{-2}\log\epsilon^{-1})$	Theorem 1.2
$1+\epsilon$	k-MSSP	$O(n^{1.529}+kn)\cdot O(\epsilon^{-1})^{\sqrt{2\log_{1/\epsilon} n}}$	Theorem 1.5
$1+\epsilon$	APSP	$O(n^2)\cdot O(\epsilon^{-1})^{\sqrt{2\log_{1/\epsilon} n}}$	Theorem 1.5
$(1+\epsilon, n^{o(1)})$	Emulators	$O(n^{1.407}\epsilon^{-2}\log\epsilon^{-1})$	Lemma 1.4

TABLE I
SUMMARY OF OUR DETERMINISTIC RESULTS FOR DISTANCE AND EMULATORS. BY k-MSSP WE MEAN MULTI-SOURCE DISTANCES FROM k SOURCES. FOR THE EXACT DEPENDENCE ON ω, SEE THE RESPECTIVE THEOREMS.

Deterministic $(1+\epsilon)$-st distances: Our main result is a deterministic, fully dynamic algorithm for maintaining a $(1+\epsilon)$-approximation of the distance between a fixed pair of nodes $s, t \in V$ whose worst-case update time matches a conditional lower bound.

Theorem 1.1: Given an unweighted undirected graph $G=(V,E)$ and a pair of nodes s and t, there is a fully-dynamic data structure for maintaining $(1+\epsilon)$-distances between s and t deterministically with

- Preprocessing time of $O(n^{\omega}\epsilon^{-2}\log\epsilon^{-1})$, where $\omega \leq 2.373$.
- Worst-case update time of $O((n^{\omega(1,1,\mu)-\mu}+n^{\omega(1,\mu,\nu)-\nu}+n^{\mu+\nu}+n^{4/3})\epsilon^{-2}\log\epsilon^{-1})$ for any parameters $0\leq\nu\leq\mu\leq 1$, which is $O(n^{1.407}\epsilon^{-2}\log\epsilon^{-1})$ for current ω ($\mu\approx 0.856$, $\nu\approx 0.551$).

We are not aware of any non-trivial deterministic algorithms with worst-case update time for maintaining the exact or $(1+\epsilon)$-approximate st-distance under both insertions and deletions.[4]

When relaxing determinism to randomization against adaptive adversaries, the previously fastest fully-dynamic algorithm for st-distance has worst-case update time $O(n^{1.724})$ [12], [14] and maintains the distance exactly for unweighted directed graphs. We later also show that if randomization is allowed, our approach also improves the bound for *exact* st-distances to $O(n^{1.7035})$.

The previously fastest fully dynamic for unweighted, undirected graphs is implied by the approach of [16] and yields a worst-case update time of $O(n^{1.529})$; this algorithm employs randomization against oblivious adversaries, and in addition to the approximate distance can also report an st-path of the corresponding length. Despite being deterministic, our algorithm improves upon these upper bounds.

Moreover, for *current bounds* on ω, our result closes the gap between previous upper bounds and a conditional lower bound for $(1+\epsilon)$-approximate dynamic st-distances on unweighted undirected graphs [12]. This conditional lower bound is based on a hardness assumption called "uMv-hinted uMv" where a vector-matrix-vector product must be computed after receiving hints about the structure of the three inputs. This assumption

[2]Two $n\times n$ matrices can be multiplied in $O(n^{\omega})$ operations with $\omega\leq 2.373$ [8]–[10]. We write $O(n^{\omega(a,b,c)})$ for the complexity of multiplying an $n^a\times n^b$ by $n^b\times n^c$ matrix [11].

[3]By a "distance oracle", we mean a data structure that supports fast queries. Our goal – unlike many static algorithms – is not optimizing the space of this data structure.

[4]In independent work, [18] obtained such deterministic bounds for (the more general) directed graphs when restricting to only edge insertions.

formalizes the current barrier for improving upon algorithms for various fully dynamic problems such as directed st-reachability, maximum matching size, directed cycle detection, directed k-cycle and k-path detection, and on the algebraic side, maintaining determinant and rank of a dynamic matrix. All of these problems admit an $O(\min_{0\leq\nu\leq\mu\leq 1}(n^{\omega(1,1,\mu)-\mu}+n^{\omega(1,\mu,\nu)-\nu}+n^{\mu+\nu}))$ worst-case update time (which for current ω amounts to $O(n^{1.407})$) and – assuming hardness of "uMv-hinted uMv" – no dynamic algorithm for these problems can improve upon this by a polynomial factor. While the nature of conditional lower bounds can never rule out the existence of faster algorithms with certainty, we believe that these connections provide evidence that a substantial breakthrough will be necessary in order to improve upon the update time of our algorithm. We further note that closing the update-time gap between fully dynamic st-reachability and st-distance was raised as an important open problem by Sankowski [19]; our bound for $(1+\epsilon)$-approximate st-distance in undirected graphs partially resolves this question.

Deterministic $(1+\epsilon)$-SSSP: Our second result is a deterministic, fully dynamic algorithm for maintaining $(1+\epsilon)$-single source distances whose worst-case update time matches a conditional lower bound.

Formally we show the following.

Theorem 1.2: Given an unweighted undirected graph $G=(V,E)$ and a single source s, and $0<\epsilon<1$, there is a deterministic fully-dynamic data structure for maintaining $(1+\epsilon)$-distances from s with worst-case update time of $O((n^{\omega(1,1,\mu)-\mu}+n^{1+\mu})\epsilon^{-2}\log\epsilon^{-1})$ for any $0\leq\mu\leq 1$. For current bounds on ω and the best choice of $\mu\approx 0.529$, this is $O(n^{1.529}\epsilon^{-2})$. The algorithm has preprocessing time of $O(n^{\omega}\epsilon^{-2}\log\epsilon^{-1})$, where $\omega\leq 2.373$.

As with the st case, we are not aware of any non-trivial deterministic algorithms with worst-case update time for maintaining exact or $(1+\epsilon)$-approximate SSSP under both insertions and deletions. When relaxing determinism to randomization against adaptive adversaries, the previously fastest fully-dynamic algorithm for $(1+\epsilon)$-approximate SSSP in unweighted graphs has a much slower worst-case update time of $O(n^{1.823})$ [20] (albeit that bound also holds for directed weighted graphs).

The fastest fully dynamic algorithm for unweighted, undirected graphs is implied by the approach of [16] and yields a worst-case update time of $O(n^{1.529})$; this algorithm employs randomization against oblivious adversaries, and in addition to the approximate distance can also report an st-path of the corresponding length. Our result matches the update time of the distance maintenance problem with a deterministic algorithm and additionally improves the dependence on the error parameter ϵ from $(1/\epsilon)^{O(\sqrt{\log_{1/\epsilon} n})}$ to a small polynomial. Moreover, this update time matches a conditional lower bound stated in [12] based on the hardness assumption "Mv-hinted Mv". This is a similar type of hardness assumption as discusses in the st-case, but tuned to single source problems. We emphasize again that our approximate st result matches the conditional lower for *current* ω, whereas our approximate SSSP bound matches the conditional lower bound for any ω.

Deterministic Sparse Emulators: The main tool developed and applied in this paper is a novel fully dynamic algorithm for maintaining $(1+\epsilon,\beta)$-emulators with various trade-offs, which might be of independent interest.

Definition 1.3: Given a graph $G=(V,E)$, an (α,β)-*emulator* of G is a graph $H=(V,E')$ (that is not necessarily a subgraph of G and might be weighted) in which $d_G(u,v)\leq d_H(u,v)\leq\alpha\cdot d_G(u,v)+\beta$ for all pairs of nodes $u,v\in V$. If H is a subgraph of G, then H is an (α,β)-*spanner* of G.

In this paper, we are mainly interested in so-called *near-additive* emulators and spanners, as introduced by [21], for which $\alpha=1+\epsilon$ for any parameter $\epsilon>0$ and β is a function of ϵ. A influential construction of Thorup and Zwick [22] gives $(1+\epsilon,\beta)$-spanners of size $\tilde{O}(n^{1+1/k})$[5] and with $\beta=O(1/\epsilon)^k$ for any $0<\epsilon\leq 1$ and $2\leq k\leq\log n$ that can statically be computed in time $\tilde{O}(mn^{1/k})$.[6] For any constant ϵ, this allows for a $(1+\epsilon,n^{o(1)})$-spanner of size $n^{1+o(1)}$.

In this paper, we obtain the following result for maintaining near-additive emulators.

Lemma 1.4: Given an unweighted, undirected graph $G=(V,E)$, parameters $0<\epsilon<1$ and $2\leq k\leq\log n$, we can maintain a $(1+\epsilon,\beta)$-emulator of G with size $\tilde{O}(n^{1+1/k})$, where $\beta=O(1/\epsilon)^k$ deterministically with worst-case update time of $\max(\tilde{O}(n^{4/3+1/k}),O(n^{1.407}\epsilon^{-2}\log\epsilon^{-1}))$. Here the latter term of the update time has the same dependence on ω as Theorem 1.1. The preprocessing time of this algorithm is $O(n^{\omega}\epsilon^{-2}\log\epsilon^{-1})$.

This result should mainly be compared to the fully dynamic algorithm of [16] for maintaining a $(1+\epsilon,n^{o(1)})$-spanner of size $n^{1+o(1)}$ with worst-case update time $O(n^{1.529})$ for any constant $\epsilon>0$ that employs randomization against an oblivious adversary. We improve upon the result of [16] both in running time and by having a deterministic algorithm at the cost of maintaining emulators instead of spanners. Other works on maintaining spanners or emulators give a multiplicative stretch $\alpha\geq 3$ [24]–[30] or are restricted to a partially dynamic setting [31], [32].

Deterministic $(1+\epsilon)$-MSSP: Another implication of our techniques is an algorithm for $(1+\epsilon)$-multi-source distances. In the full version, we give an algorithm that combines our sparse emulator construction with the algebraic techniques to prove the following theorem.

Theorem 1.5: Given an unweighted, undirected graph $G=(V,E)$, and $0<\epsilon<1$, and a fixed set of sources S, we can maintain $(1+\epsilon)$-approximate distances from S (i.e. pairs in $S\times V$) deterministically with $O((n^{\omega(1,1,\mu)-\mu}+n^{1+\mu}+|S|\cdot n)\cdot O(1/\epsilon)^{\sqrt{2\log_{1/\epsilon} n}}$ worst-case update time, which for current ω

[5]Throughout this paper, we use $\tilde{O}(\cdot)$-notation to suppress terms that are polylogarithmic in n, the number of nodes of the graph.

[6]In static settings there are somewhat more involved algorithms for near-additive emulators that lead to slightly better tradeoffs in specific parameter settings (e.g. see [23]).

is $O(n^{1.529} + |S| \cdot n) \cdot O(1/\epsilon)^{\sqrt{2\log_{1/\epsilon} n}}$. The preprocessing time is $O(n^\omega) \cdot O(\frac{1}{\epsilon})^{\sqrt{2\log_{1/\epsilon} n}}$.

Hence we can maintain distances from up to $\tilde{O}(n^{0.52})$ sources in almost (up to an $n^{o(1)}$ factor) the same time as maintaining distances from a single-source.

Deterministic $(1+\epsilon)$-APSP: One implication of Theorem 1.5 (by simply setting $S = V$) is a deterministic fully-dynamic algorithm for maintaining all-pairs-shortest path that nearly (up to an $n^{o(1)}$ factor) matches the trivial lower bound of $\Omega(n^2)$ time per update for this problem. More formally, Given an unweighted, undirected graph $G = (V, E)$, and $0 < \epsilon < 1$, we can maintain $(1+\epsilon)$-all-pairs distances deterministically with $O(n^2) \cdot O(\frac{1}{\epsilon})^{\sqrt{2\log_{1/\epsilon} n}}$ worst-case update time. The preprocessing time is $O(n^\omega) \cdot O(\frac{1}{\epsilon})^{\sqrt{2\log_{1/\epsilon} n}}$.

It is worth mentioning that there is another (simpler) approach to obtain this bound that we will discuss in the full version. The previous comparable bounds for this problem either used randomization [20] or have amortized bounds [33], [34]. The fastest deterministic algorithm with worst-case guarantee that maintains exact shortest paths unweighted, directed graphs and has an update time of $\tilde{O}(n^{2.6})$ [35].

B. Randomized Algorithms for Exact st-distance, diameter approximation and $(1+\epsilon)$-APSP Distance Oracles

In the full-version of this paper we show several results that, unlike our previous bounds, are randomized. This includes an improved bound for exact st-distances using our new algebraic data structures, and two other implications of our dynamic multi-source algorithms.

Exact st-distances: Our new dynamic algorithm for maintaining bounded distances also leads to improved bounds for dynamic exact st-distances in *directed graphs*, if we allow randomization. For current ω, we obtain a worst-case update time of $O(n^{1.7035})$, improving upon the previous best bound of $O(n^{1.7643})$ [12], [14].

Theorem 1.6: For any $0 \leq \nu \leq \mu \leq 1$ and $0 \leq h \leq n$, there exists a randomized dynamic algorithm that maintains exact st-distances in directed graphs. The preprocessing time is $\tilde{O}(hn^\omega)$ and the worst-case update time per edge insertion or deletion is $\tilde{O}(h(n^{\omega(1,1,\mu)-\mu} + n^{\omega(1,\mu,\nu)-\nu} + n^{\mu+\nu} + (n/h)^2))$. After each update, the algorithm returns the exact st-distance and the result is correct with high probability. The algorithm works against an adaptive adversary.

For current bounds on ω, this is $O(n^{1.7035})$ time per update (with $\mu \approx 0.8556$, $\nu \approx 0.5512$ and $\log_n(h) \approx 0.2966$).

Randomized Approximate Diameter: We can maintain a nearly-$(3/2+\epsilon)$-approximation of the diameter in fully dynamic unweighted graphs in $O(n^{1.596}) \cdot (\frac{1}{\epsilon})^{O(1)}$ worst-case update time against an adaptive adversary. This is done by using our emulator to compute $(1+\epsilon)$-MSSP algorithms for certain sets S of size $\tilde{O}(\sqrt{n})$ based on an algorithm by [36].

Previously, the fastest fully-dynamic algorithm with this approximation guarantee by [20] had a worst-case update time of $O(n^{1.779})$ and employed randomization against an adaptive adversary. We get better bounds by combining our sparse emulator algorithms with the algorithm of [20].

Dynamic diameter was also analyzed in the partially dynamic setting [37], [38], e.g. there exists a nearly-$(3/2+\epsilon)$-approximate decremental algorithm with $m^{1+o(1/\epsilon)}\sqrt{n}/\epsilon^2$ expected total update time [37].

$(1+\epsilon)$-APSP Distance Oracles with Sublinear Query: Another implication of our new approach for dynamic $(1+\epsilon)$-MSSP is an improved bound for maintaining a data structure supporting all-pairs distance queries that has subquadratic update time $O(n^{1.788}) \cdot O(\frac{1}{\epsilon})^{\sqrt{2\log_{1/\epsilon} n}}$ and a small polynomial query time $O(n^{0.45}\epsilon^{-2})$ against an adaptive adversary.

Our result directly improves upon the $O(n^{1.862}\epsilon^{-2}\log\epsilon^{-1})$ update time of a corresponding algorithm by [20] which has the same query time as ours and also employs randomization against an adaptive adversary. The algorithm of [20] internally maintains $(1+\epsilon)$-approximate MSSP and thus our result is almost directly implied by our improvement for maintaining approximate MSSP.

C. Further Related Work

Several state-of-the art dynamic algorithms employ an algebraic approach (i.e. use fast matrix multiplication) for maintaining reachability and distance information. As a conditional lower bound by Abboud and Vassilevska Williams [39] shows, this is inherent in certain regimes: Unless one is able to multiply two $n \times n$ boolean matrices in $O(n^{3-\delta})$ time for some constant $\delta > 0$, no fully dynamic algorithm for st reachability in directed graphs can beat $O(n^{2-\delta'})$ update and query time and $O(n^{3-\delta'})$ preprocessing time (for some constant $\delta' > 0$).

While not explicitly stated in [39], the same conditional lower bound extends to fully dynamic $(1+\epsilon)$-approximate st distances on undirected unweighted graphs for a small enough constant ϵ.

In the same spirit, [40] obtained a more refined conditional lower bound for combinatorial algorithms maintaining sparse near-additive spanners and emulators based on the Combinatorial k-Clique hypothesis.

The use of algebraic techniques for maintaining reachability and distance information can be traced back to the path counting approaches of King and Sagert [41] and Demetrescu and Italiano [42]. Sankowski [13] subsequently developed a more general framework for maintaining the adjoint of a matrix and applied it to maintaining reachability in directed graphs [13] and distances in unweighted, directed graph [14]. This approach was further refined which led to improved dynamic algorithms for reachability [12] as well as for approximate distances [20]. Recently, such algebraic data structures have been enriched to maintain "witnesses" that allow reporting paths in addition to the pure reachability/distance information: the path reporting mechanism of [16] uses randomization against an oblivious adversary and the one of [18] uses randomization against an adaptive adversary. The latter paper also contains deterministic bounds for incremental approximate shortest paths independently of our work.

II. Technical Overview

In this section we give a high-level overview of our technical contributions. In Section II-A, we start by presenting deterministic algorithms for maintaining $(1+\epsilon, 2)$ and $(1+\epsilon, 4)$-emulators with applications respectively in $(1+\epsilon)$-SSSP and $(1+\epsilon)$-st distances. These emulator algorithms slightly extend a known "localization" [32] of the (randomized) additive emulator construction [43] and have two properties crucial for our bounds: (1) They are based on a "deterministic" and "slowly changing" hitting set of high-degree neighborhoods. (2) For assigning the edge weights, we only need to compute bounded pairwise distances between the smaller set of nodes involving the hitting set. We show that in our setting we can – instead of using a standard randomized approach – deterministically maintain an approximate solution to this particular hitting-set instance with low recourse.

In Section II-B, we then design an algebraic data structure for maintaining bounded distances in such a way that it can deal with a gradually changing hitting set efficiently. Following the approach by Sankowski [14], maintaining small distances between certain vertices reduces to maintaining a submatrix of some dynamic matrix inverse. We modify the dynamic matrix inverse algorithm of [12] to efficiently maintain such a submatrix. In general, the algorithm of [12] has faster update but slower query time compared to other dynamic matrix inverse algorithms [13]. However, by exploiting that the queries will be located within some specified submatrix, we can speed up the query complexity. Using this additional information about the location of the queries, we can periodically precompute larger batches of information during the update phase via fast matrix multiplication. For getting this speed up we need to modify the algorithm and analysis of [12], as their algorithm has different layers that need to be handled separately in our case.

Finally, in Section II-C we discuss how using further resparsifications we can obtain near linear size additive spanners with applications in MSSP, APSP, and diameter approximation.

A. Dynamic Emulators via Low-Recourse Hitting Sets

Deterministic $(1+\epsilon, 2)$-emulator and $(1+\epsilon)$-SSSP: We start with a deterministic algorithm for maintaining a $(1+\epsilon, 2)$-emulator. This algorithm is inspired by a randomized algorithm (working against an oblivious adversary) used by [32] in the decremental setting, which in turn is based on the purely additive static construction of [43]. Given an unweighted graph $G = (V, E)$, we maintain an emulator H with size $\tilde{O}(n^{3/2})$ as follows:

1) Let $d = \sqrt{n}$ be a degree threshold. For any node v where $\deg(v) < d$, add all the edges incident to v to H. These edges have weight 1.
2) Construct a *hitting set* $A \subseteq V$ of size $\tilde{O}(\sqrt{n})$, such that every node with degree at least d, called a *heavy node*, has a neighbor in A. Add an edge to this neighbor.
3) For any node $u \in A$, add an edge to all nodes within distance $\lceil 2/\epsilon \rceil + 1$ to u. Set the weight of such an edge (u, w) to $d_G(u, w)$.

It is easy to see that if we were interested in a randomized algorithm that only works against an oblivious adversary, we could simply construct a hitting set A by uniformly sampling a fixed set of size $\tilde{O}(\sqrt{n})$ [44].

We could then maintain the corresponding $(\lceil 2/\epsilon \rceil + 1)$-bounded distances for all pairs in $A \times V$ after each update using the algebraic data structure by [14] which runs in $O(n^{1.529}\epsilon^{-1}\log\epsilon^{-1})$ time per update.

The distance bound of $(\lceil 2/\epsilon \rceil + 1)$ in our emulator algorithms leverages the power of algebraic distance maintenance data structures because their running times scale with the given distance bound. However, these ideas alone are not enough for obtaining an efficient deterministic algorithm. We will have to change both the hitting set construction and the algebraic data structure.

Before explaining how to maintain both the hitting set and the corresponding distances deterministically, let us sketch the properties of this emulator and how it can be used for maintaining $(1+\epsilon)$-SSSP. It is easy to see that H has size $\tilde{O}(n^{3/2})$: we add $\tilde{O}(nd)$ edges incident to low-degree nodes, and $\tilde{O}(n^{3/2})$ edges in $A \times V$. For the stretch analysis, consider any pair of nodes s, t, and let π be the shortest path between s, t. We can divide π into segments of equal length $\lceil 2/\epsilon \rceil$, and possibly one additional smaller segment. Consider one such segment $[u, v]$. If all the nodes on this segment are low-degree, then we have included all the corresponding edges in the emulator. Otherwise there is a node $w \in A$ that is adjacent to the first heavy node on this segment. We have $d_G(w, v) \leq \lceil 2/\epsilon \rceil$, and thus in the third step of the algorithm we have added a (weighted) edge (w, v) in the emulator. It is easy to see that the path going through w either provides a $(1+\epsilon)$ multiplicative factor, or (for the one smaller segment) an additive term of 2.

Given a $(1+\frac{\epsilon}{2}, 2)$ emulator, we can now maintain $(1+\epsilon)$-SSSP by (i) using algebraic techniques to maintain $O(1/\epsilon)$-bounded distances from the source s to all nodes in V, and (ii) statically running Dijkstra's algorithm on the emulator in time $\tilde{O}(n^{3/2})$, and finally (iii) taking the minimum of the two distance values for each pair $(s, v) \in \{s\} \times V$. We observe that if $d_G(s, v) \leq O(\frac{1}{\epsilon})$, then we are maintaining a correct estimate in step (i). Otherwise in step (ii) the combination of the $(1+\frac{\epsilon}{2})$ multiplicative factor and the additive term, leads to an overall $(1+\epsilon)$-approximate estimate.

Deterministic low-recourse hitting set: As discussed, we can easily obtain a fixed hitting set of size $\tilde{O}(n/d)$ using randomization, but we are interested in a deterministic algorithm. One natural approach for constructing the hitting set A deterministically is as follows: For each node v with degree at least d, consider a set of exactly d neighbors of v. After *each update* we can *statically* and *deterministically* compute an $O(\log n)$-approximation to this instance of the hitting set problem. We use a simple greedy algorithm that proceeds by sequentially adding nodes to A that hit the maximum number of uncovered heavy nodes.

This can be done in $\tilde{O}(nd)$ time and gives us a hitting set of size $\tilde{O}(n/d)$ as well. This running time is within our

desired update-time bound, but we also need to maintain $\Theta(1/\epsilon)$-bounded distances from elements in this hitting set. As we outline in the full version, by using the naive approach of recomputing a hitting set in each update and employing off-the-shelf algebraic data structures (e.g. [12], [14]) for maintaining bounded distances in $A \times V$, we would get an update time of $O(n^{1.596})$ for current ω. However, there is a conditional lower bound of $O(n^{1.529})$ for this problem [12], and our goal is to design an algorithm that matches this bound.

To get a better running time, we change both our construction and the algebraic data structure to use a *low-recourse hitting set* instead, which ensures that in each update only a constant number of nodes are added to the set. More formally in the full version we prove the following lemma (the details can be found in the full version):

Lemma 2.1: Given a graph $G = (V, E)$ undergoing edge insertions and edge deletions and a degree threshold d, call a node v heavy if it has degree at least d. We can deterministically maintain a hitting set A_d of size $O(\frac{n \cdot \log n}{d})$ with worst-case $O(1)$ recourse and worst-case $O(d^2 + d \log n)$ time per update (after $O(nd)$ preprocessing time) such that all heavy nodes have a neighbor in A_d.

At a high-level our dynamic low recourse hitting set proceeds as follows: we start by using the static greedy hitting set algorithm. We then note that each update (insertion or deletion) can make at most 2 heavy nodes uncovered. We can keep on adding arbitrary neighbors of such nodes to our hitting set A until the size of the hitting set exceeds its initial $O(\frac{n}{d} \log n)$ bound by a constant factor, and then reset the construction. This leads to an amortized constant recourse bound, and we can then use standard techniques to turn this into a worst case constant recourse bound (see the full version for the full algorithm).

Note that this hitting set problem can be seen as a set cover instance of size $O(nd)$, where each set consists of exactly d neighbors of a heavy node. Dynamic set cover approximation has received significant attention in recent years (e.g. [40], [45]–[47]). The most relevant result to our setting is a fully-dynamic $O(\log n)$-approximate set cover algorithm by [47]). However we cannot use their result directly, as they state that their polynomial time algorithm only leads to constant *amortized* recourse, and their update-time guarantees are also only amortized[7]. Here we use a simple approach that utilizes the properties of our hitting set instance, which is enough to get *worst-case* recourse bounds.

Deterministic $(1+\epsilon, 4)$-emulator for $(1+\epsilon)$-st distances: Next, we outline how we can improve the $O(n^{1.529})$ update time to $O(n^{1.407})$ in case of st-distances. For this purpose, we maintain a $(1 + \epsilon, 4)$-emulator with size $\tilde{O}(n^{4/3})$, which again is inspired by the purely additive construction of [43] in the static setting, by making the following modifications to the algorithm described in Section II-A above: We set the degree threshold to $d = n^{1/3}$. More importantly, rather than adding edges corresponding to bounded distances in $A_d \times V$, we only add pairwise edges between nodes (with bounded distance) in $A_d \times A_d$. This has two advantages: First, we can run Dijkstra on a sparser graph. Second, the algebraic steps can be performed much faster when we only need to maintain pairwise distances between two sets of sublinear size (here $|A_d| = \tilde{O}(n^{2/3})$, rather than from a set of size $\tilde{O}(\sqrt{n})$ to *all* nodes in V.

It is easy to see that this emulator has size $\tilde{O}(n^{4/3})$. There are $\tilde{O}(nd)$ edges corresponding to low-degree nodes, and $\tilde{O}(n^{4/3})$ corresponding to edges in $A_d \times A_d$. The stretch argument follows a similar structure to the one for the $(1 + \epsilon, 2)$-emulator. Again, for each pair of nodes s, t, we divide the shortest path to segments of equal length $\Theta(1/\epsilon)$. The main difference is that here we should consider the first and last heavy nodes on each segment, which we denote by x and y. Then there must be nodes $w_1, w_2 \in A_d$ that are adjacent to x and y respectively. We have $d_G(w_1, w_2) \leq \Theta(1/\epsilon)$ and thus we have added an edge (w_1, w_2) in the emulator. The path using this edge will lead to either a $(1+\epsilon)$-multiplicative stretch for this segment, or an additive term of 4 for the (at most) one smaller segment.

Note that this algorithm does not lead to better bounds for single-source distances since querying $\Theta(1/\epsilon)$-bounded single-source distances still takes $O(n^{1.529})$ time using known algebraic techniques. However, if we are interested in the $\Theta(1/\epsilon)$-bounded distance between a fixed pair of nodes s and t, our algebraic approach, as outlined in Section II-B, leads to better bounds. In this case, we get an improved bound of $O(n^{1.407})$.

B. Dynamic Pairwise Bounded Distances via Matrix Inverse

As outlined before, we must efficiently maintain bounded pairwise distances for some sets $S \times T \subseteq V \times V$, where the sets S and T are dynamically changing. We additionally use the fact that even though these sets change, they do not change substantially with each update because of our low-recourse hitting sets. In this section, we outline the following: (i) a reduction from maintaining $S \times T$-distances to maintaining a submatrix[8] $(\mathbf{A}^{-1})_{S,T}$ for some dynamic matrix $\mathbf{A}$, and (ii) a dynamic algorithm maintaining this submatrix of the inverse efficiently. This dynamic matrix inverse algorithm, together with the reduction, then imply the following dynamic algorithm (Theorem 2.2, proven in the full version) for maintaining bounded distances.

Theorem 2.2: For all $0 \leq \nu \leq \mu \leq 1$ there exists a deterministic dynamic algorithm that, after preprocessing a given unweighted directed graph G and sets $S, T \subseteq V$, supports edge-updates to G and set-updates to S and T (i.e. adding or removing a node to S or T) as long as $|S|, |T| \leq n^{\mu}$ throughout all updates. After each edge- or set-

[7]The goal in [47] is a generic set cover approximation algorithm, which is why they are not comparable to our specialized algorithm. Also, the other set cover algorithms cited lead to approximation ratio dependent on an instance parameter f, which can be as large as n in our case.

[8]Throughout, we use $\mathbf{N}_{S,T}$ for sets $S, T \subseteq [n]$ and $n \times n$ matrix $\mathbf{N}$ to denote the submatrix consisting of rows with index in S and columns with index in T.

update the algorithm returns the h-bounded pairwise distances of $S \times T$ in G.

The preprocessing time is $O(n^\omega h^2 \log h)$, and the worst-case update time is

$$O((n^{\omega(1,1,\mu)-\mu} + n^{\omega(1,\mu,\nu)-\nu} + n^{\mu+\nu} + |S \times T|)h^2 \log h).$$

For current bounds on rectangular matrix multiplication $\omega(\cdot,\cdot,\cdot)$ [11], this is $O((n^{1.407} + |S \times T|)h^2 \log h)$ for $|S|, |T| \leq n^{0.85}$, or $O((n^{1.529} + |S \times T|)h^2 \log h)$ for any (possibly larger) S, T.

For our approximate st-distance algorithm, we will set $|S| = |T| = \tilde{O}(n^{2/3})$ and $h = O(1/\epsilon)$, resulting in $O(n^{1.407}\epsilon^{-2} \log \epsilon^{-1})$ update time. For our approximate SSSP algorithm we will set $|S| = n$, $|T| = \tilde{O}(\sqrt{n})$ and $h = O(1/\epsilon)$, resulting in $O(n^{1.529}\epsilon^{-2} \log \epsilon^{-1})$ update time.

Reducing distances to matrix inverse: All previous fully dynamic algebraic algorithms that maintain distances work by reducing the task to the so called "dynamic matrix inverse" problem [12], [14], [16], [20], [48], [49]. This reduction is due to Sankowski [14] who originally used the adjoint instead of the matrix inverse. In previous work on fully dynamic algebraic algorithms, this reduction was always randomized. Here we recap the reduction when using matrix inverse instead of adjoint, and argue why the reduction can be derandomized for our use-case of maintaining bounded distances. Readers already familiar with this reduction might want to skip ahead to the paragraph labeled "*Submatrix maintenance*".

For the reduction, we are given an adjacency matrix $\mathbf{A}$. Note that $\mathbf{A}^k_{s,t}$ (where $\mathbf{A}^k$ is the k-th power of $\mathbf{A}$) is the number of (not necessarily simple) paths from s to t of length k. Specifically, the smallest k with $\mathbf{A}^k_{s,t} \neq 0$ is the distance from s to t. We can maintain these powers of $\mathbf{A}$ via dynamic matrix inverse as follows:

Let X be some symbol and let $(\mathbf{I} - X\mathbf{A})$ be the matrix with 1 on the diagonal and $(\mathbf{I} - X\mathbf{A})_{u,v} = -X$ for all edges $(u, v) \in E$. When performing all arithmetic operations[9] modulo X^h, we have $(\mathbf{I} - X\mathbf{A})^{-1} = \sum_{k=0}^{h-1} X^k \mathbf{A}^k$. To see this, observe

$$(\mathbf{I} - X\mathbf{A}) \cdot \sum_{k=0}^{h-1} X^k \mathbf{A}^k = \sum_{k=0}^{h-1} X^k \mathbf{A}^k - \sum_{k=1}^{h} X^k \mathbf{A}^k = \mathbf{I}$$

where the last identity holds by $X^h \mathbf{A}^k = 0$ because of the entry-wise mod X^h. Thus, a dynamic algorithm that maintains the inverse of matrix $(\mathbf{I} - X\mathbf{A})$ is able to maintain distances of length $< h$ in dynamic graphs. The task of maintaining pairwise distances for $S \times T$ thus reduces to the task of maintaining the submatrix $(\mathbf{M}^{-1})_{S,T}$ for some dynamic matrix $\mathbf{M}$.

Note that the number of uv-paths of length k, given by $\mathbf{A}^k_{u,v}$, might be as large as $O(n^k)$. Representing this number needs $O(k)$ words in Word-RAM model and each arithmetic operation needs $O(k)$ time [50]. In general, a graph might have paths of length $O(n)$, thus randomization was used in previous work [12], [14], [16], [20], [48], [49] to bound the bit-length and arithmetic complexity of the numbers involved (e.g. by maintaining the number of paths modulo some small random prime $p = \text{poly}(n)$, or by using Schwartz-Zippel lemma).[10]

However, in our use-case, we only need distances up to $O(1/\epsilon)$ thanks to properties of our emulators, thus the randomization is not required. Each arithmetic operation will only need $O(1/\epsilon)$ time as we only consider numbers represented by $O(1/\epsilon)$ words.

Submatrix maintenance: As explained in the previous paragraph, our dynamic distance algorithms reduce to a dynamic matrix inverse algorithm that maintains a submatrix $\mathbf{M}^{-1}_{S,T}$ for some dynamic matrix $\mathbf{M}$. Any existing dynamic matrix algorithm can maintain such a submatrix by just querying all $|S \times T|$ entries after each change to $\mathbf{M}$, but this would not be fast enough for our purposes. We instead propose a new dynamic matrix inverse algorithm that can maintain such a submatrix efficiently, if the sets S and T are slowly changing.

The construction of this dynamic algorithm relies on reducing maintaining $\mathbf{M}^{-1}_{S,T}$ to maintaining partial rows of the form $(\mathbf{M}^{-1})_{k,T}$ for any $k \in [n]$, which is formalized and proven in the full version.

Lemma 2.3: Assume we are given a dynamic algorithm that initializes on a dynamic set $T \subset [n]$ and a dynamic $n \times n$ matrix $\mathbf{M}$ that is promised to stay non-singular. Assume the algorithms supports both changing any entry of $\mathbf{M}$ and adding/removing any index to/from T in $O(u(|T|, n))$ operations, and supports queries for any $i \in [n]$ that return $\mathbf{M}^{-1}_{i,T}$ in $O(q(|T|, n))$ operations. Then we can also maintain $\mathbf{M}^{-1}_{S,T}$ explicitly for dynamic matrix $\mathbf{M}$ and dynamic sets $S, T \subset [n]$ while the update time increases to $O(u(k, n) + q(k, n) + |S \times T|)$ for $k = \max(|S|, |T|)$. The preprocessing time increases by an additive $O(n^\omega)$ operations.

Thus it suffices to design a dynamic matrix inverse algorithm that supports efficient queries to partial rows $\mathbf{M}^{-1}_{i,T}$. Our proposed algorithm is a modification of the dynamic matrix inverse algorithm by [12]. Their data structure has the fastest known update complexity among all dynamic matrix inverse algorithms, but comes at the cost of slower queries than some data structures from [13].

We speed up the queries of [12] by exploiting the fact that set T is slowly changing, thus we know ahead of time which entries of the inverse might be queried in the future. By preprocessing these entries, we can speed up queries to $\mathbf{M}^{-1}_{i,T}$ for any $i \in [n]$ and a dynamic set $T \subset [n]$. In addition to these faster queries, we also simplify the proof and the structure of the dynamic algorithm from [12].

We next explain how to achieve such a speed up. We start with a quick recap of how the data structure of [12] represents the dynamic matrix inverse and then explain how we modify the algorithm. Let $\mathbf{M}'$ be the dynamic matrix $\mathbf{M}$

[9]For our proofs, this is formalized as the entries of the matrix being from $\mathbb{F}[X]/\langle X^h \rangle$ for some field $\mathbb{F}$, i.e. polynomials over $\mathbb{F}$ where we truncate all monomials of degree $\geq h$.

[10]We focus on fully dynamic algorithms here. We note that in the *incremental* setting (i.e. only edge insertions), such randomization is not required. See e.g. [18].

during initialization, then we maintain $\mathbf{M}$ in the following implicit form:

$$\mathbf{M} = \mathbf{M}' + \mathbf{U}'\mathbf{V}'^\top + \mathbf{U}\mathbf{V}^\top \tag{1}$$

where for some $0 \le \nu \le \mu \le 1$, the matrices $\mathbf{U}', \mathbf{V}'$ have at most n^μ columns and $\mathbf{U}, \mathbf{V}$ have at most n^ν columns, all of which have at most one non-zero entry per column. Initially, $\mathbf{U}, \mathbf{U}', \mathbf{V}, \mathbf{V}'$ are all empty matrices (i.e. with 0 columns) as $\mathbf{M} = \mathbf{M}'$. Then, with each update to $\mathbf{M}$, we update $\mathbf{U}$ and $\mathbf{V}$ as follows: The entry update to $\mathbf{M}_{i,j}$ can be represented as adding some $v \cdot e_i e_j^\top$ to $\mathbf{M}$ for some scalar v. We can thus maintain (1) by setting $\mathbf{U} \leftarrow [\mathbf{U}|v \cdot e_i]$ and $\mathbf{V} \leftarrow [\mathbf{V}|e_j]$ (i.e. appending a new column to $\mathbf{U}$ and $\mathbf{V}$). After n^ν updates, the matrices $\mathbf{U}$ and $\mathbf{V}$ have n^ν columns and we append these columns to $\mathbf{U}', \mathbf{V}'$ by setting $\mathbf{U}' \leftarrow [\mathbf{U}'|\mathbf{U}]$, $\mathbf{V}' \leftarrow [\mathbf{V}'|\mathbf{V}]$, then we reset $\mathbf{U}, \mathbf{V}$ to be empty matrices (i.e. with 0 columns). Thus $\mathbf{M}$ is still maintained in form (1) and we can assume $\mathbf{U}$, $\mathbf{V}$ always have at most n^ν columns. After n^μ updates, the algorithm is reset by letting $\mathbf{M}' \leftarrow \mathbf{M}$ and all $\mathbf{U}, \mathbf{U}', \mathbf{V}, \mathbf{V}'$ are reset to be empty matrices. Thus we can also assume $\mathbf{U}', \mathbf{V}'$ have at most n^μ columns.

The task is now to maintain $\mathbf{M}^{-1}$ in some implicit form that allows for fast queries to $\mathbf{M}^{-1}_{i,T}$ for any $i \in [n]$ and a dynamic set $T \subset [n]$. For this consider the following Sherman-Morrison-Woodbury identity.

Lemma 2.4 ([51], [52]): For any non-singular $\mathbf{M}$ and $\mathbf{M} + \mathbf{U}\mathbf{V}^\top$ we have

$$(\mathbf{M}+\mathbf{U}\mathbf{V}^\top)^{-1} = \mathbf{M}^{-1} - \mathbf{M}^{-1}\mathbf{U}(\mathbf{I}+\mathbf{V}^\top\mathbf{M}^{-1}\mathbf{U})^{-1}\mathbf{V}^\top\mathbf{M}^{-1}.$$

By applying this identity twice (once for $\mathbf{M} := \mathbf{M}'' + \mathbf{U}\mathbf{V}^\top$ and once for $\mathbf{M}'' := \mathbf{M}' + \mathbf{U}'\mathbf{V}'^\top$) we can write:

$$\begin{aligned} \mathbf{M}^{-1} = \underbrace{\mathbf{M}'^{-1} + \mathbf{A}\mathbf{B}}_{=\mathbf{M}''^{-1}} + \mathbf{M}''^{-1}\mathbf{U}\mathbf{C}\mathbf{V}^\top\mathbf{M}''^{-1} \\ \mathbf{A} := \mathbf{M}'^{-1}\mathbf{U}'(\mathbf{I} + \mathbf{V}'^\top\mathbf{M}'^{-1}\mathbf{U}')^{-1}, \\ \mathbf{B} := \mathbf{V}'^\top\mathbf{M}'^{-1}, \quad \mathbf{C} := (\mathbf{I} + \mathbf{V}^\top\mathbf{M}''^{-1}\mathbf{U})^{-1} \end{aligned} \tag{2}$$

where matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are maintained by the data structure. In [12], it was shown that this representation (that is, matrices $\mathbf{M}'^{-1}, \mathbf{A}, \mathbf{B}, \mathbf{U}, \mathbf{V}, \mathbf{C}$) can be maintained in $O(n^{\omega(1,1,\mu)-\mu} + n^{\omega(1,\mu,\nu)-\nu} + n^{\mu+\nu})$ time per update.[11]

Here the representation of $\mathbf{M}^{-1}$ and $\mathbf{M}''^{-1}$ is only implicit via (2), while matrices $\mathbf{M}'^{-1}$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{U}$, $\mathbf{V}$, $\mathbf{C}$ are known explicitly (i.e. direct read access in memory). The first row of Figure 1 shows (2) where each box represents one of the matrices and gray matrices are computed explicitly. We modify the algorithm by computing some submatrices of the implicit $\mathbf{M}''^{-1}$ and $\mathbf{V}^\top\mathbf{M}''^{-1}$ explicitly (see gray areas in the second row of Figure 1). Every time matrices $\mathbf{A}$ and $\mathbf{B}$ change (i.e. every n^ν iterations) we precompute $\mathbf{M}''^{-1}_{[n],T} = \mathbf{M}'^{-1}_{[n],T} + (\mathbf{A}\mathbf{B}^\top)_{[n],T}$ for current set T. It is possible to show that this precomputation can be performed in $O(n^{\omega(1,\mu,\nu)})$ operations if $|T| \le n^\mu$. Since T is slowly changing, whenever we attempt to query $\mathbf{M}''^{-1}_{i,T}$ at a later point, there are at most $O(n^\nu)$ entries that have not been precomputed yet. Each of these missing entries can be computed in $O(n^\mu)$ time because $\mathbf{M}''^{-1}_{i,j} = \mathbf{M}'^{-1}_{i,j} + (e_i^\top\mathbf{A})(\mathbf{B}e_j)$ where $\mathbf{A}$ and $\mathbf{B}$ have at most n^μ columns. Thus any row $\mathbf{M}''^{-1}_{i,T}$ can be obtained in $O(n^{\nu+\mu})$ operations.

With every update to $\mathbf{M}$, we also maintain the columns of $\mathbf{V}^\top\mathbf{M}''^{-1}$ with index in T. Note that by $\mathbf{V}$ having at most n^ν columns, each with only one non-zero entry, $\mathbf{V}^\top\mathbf{M}''^{-1}$ are just $\le n^\nu$ rows of $\mathbf{M}''^{-1}$. Further, with each update to $\mathbf{M}$, $\mathbf{V}$ grows by one column, so one more row of $(\mathbf{M}''^{-1})_{i,[n]}$ is added to $\mathbf{V}^\top\mathbf{M}''^{-1}$ for some $i \in [n]$. So we can maintain the desired submatrix of $\mathbf{V}^\top\mathbf{M}''^{-1}$ by querying the entries $\mathbf{M}''^{-1}_{i,T}$ in $O(n^{\mu+\nu})$ operations. If an index is added to T, we need to compute one new column of $\mathbf{V}^\top\mathbf{M}''^{-1}$, which means we just need to query $\le n^\nu$ entries of $\mathbf{M}''^{-1}$. This can also be done in $O(n^{\nu+\mu})$ operations.

With these explicit submatrices maintained (see Figure 1 for a summary), we can now query any $\mathbf{M}^{-1}_{i,T}$ efficiently as follows: Query $\mathbf{M}''^{-1}_{i,T}$ in $O(n^{\nu+\mu})$ operations, then query $(\mathbf{M}''^{-1}\mathbf{U}\mathbf{C}\mathbf{V}^\top\mathbf{M}''^{-1})_{i,T}$. For the latter, note that $e_i^\top\mathbf{M}''^{-1}\mathbf{U}$ are just n^ν entries of $\mathbf{M}''^{-1}$ because $\mathbf{U}$ has only one non-zero entry per column, and the columns with index in T of $\mathbf{V}^\top\mathbf{M}''^{-1}$ are maintained explicitly. Thus, this also takes just $O(n^{\nu+\mu})$ operations by $|T| \le n^\mu$.

In summary, our modification has amortized complexity (which can be made worst-case via standard techniques, see e.g. [12, Theorem B.1])

$$O(\underbrace{n^{\omega(1,\mu,\nu)-\nu}}_{\substack{\text{Explicitly maintain} \\ \text{submatrix of } \mathbf{M}''^{-1}.}} + \underbrace{n^{\mu+\nu}}_{\substack{\text{Explicitly maintain} \\ \text{submatrix of } \mathbf{V}''^\top\mathbf{M}''^{-1} \\ \text{+query any } \mathbf{M}''^{-1}_{i,T}}})$$

This is subsumed by the complexity of [12] for maintaining the matrices $\mathbf{M}'^{-1}, \mathbf{A}, \mathbf{B}, \mathbf{C}$ in (2). So our modification of their algorithm does not increase the update complexity despite precomputing submatrices of $\mathbf{M}''^{-1}$ and $\mathbf{V}'^\top\mathbf{M}''^{-1}$.

C. Sparse Emulators, MSSP, APSP, and Further Applications

Finally, we give another algorithm that lets us maintain much sparser emulators, which further leads to improvements when we need to maintain approximate distances from many sources (e.g. MSSP and APSP).

We start by maintaining near-linear size emulators as follows: first *maintain* a $(1+\epsilon, 4)$-emulator H_1 of G. Then statically construct a much sparser $(1+\epsilon, n^{o(1)})$-emulator H_2 of size $\tilde{O}(n^{1+o(1)})$. The key idea here is to use H_1 in order to construct H_2 more efficiently. We use a *static* deterministic emulator algorithm (based on [22], [53]) that can construct such an emulator in time $O(|E(H_1)|n^{o(1)})$. This leads to a fully-dynamic algorithm for maintaining $(1+\epsilon, n^{o(1)})$-

[11] Technically, [12] uses Lemma 2.4 in the form $(\mathbf{M} + \mathbf{U}\mathbf{V}^\top)^{-1} = \mathbf{M}^{-1}\mathbf{T}$ for $\mathbf{T} = (\mathbf{U}(\mathbf{I}+\mathbf{V}^\top\mathbf{M}^{-1}\mathbf{U})^{-1}\mathbf{V}^\top\mathbf{M}^{-1})$. Then sum (2) is written as a matrix product of two such $\mathbf{T}$, one for $\mathbf{U}\mathbf{V}^\top$ and one for $\mathbf{U}'\mathbf{V}'^\top$. In the full version we reprove the algorithm in sum-form (2) which simplifies both the analysis of the algorithm and the analysis of our modifications to accelerate the queries.

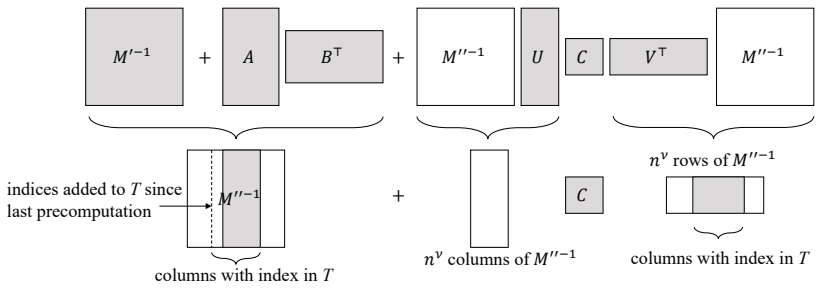

Fig. 1. Implicit representation of $\mathbf{M}^{-1}$ (2). Gray blocks represent (sub-)matrices that are explicitly maintained. White (sub-)matrices are only implicitly accessible (i.e. entries can be queried). Matrices $\mathbf{A}, \mathbf{B}$ have n^μ columns while $\mathbf{U}, \mathbf{V}, \mathbf{C}$ have n^ν columns for $0 \leq \nu \leq \mu \leq 1$. Matrices $\mathbf{U}, \mathbf{V}$ have only one non-zero entry per column.

emulators deterministically in $\tilde{O}(n^{1.407})$ worst-case update time.

Now we can use this to maintain multi-source distances from many (up to $O(n^{0.52})$) sources with an update time almost the same as the time required for single-source distances. For this purpose, given a set of sources S, we use the above approach to maintain an emulator of size $\tilde{O}(n^{1+o(1)})$. Then, similar to before, in each update we find distance estimates for pairs in $S \times V$ by computing the minimum of the following estimates: i) $n^{o(1)}$-bounded distances from all sources maintained by an algebraic data structure, ii) distances from all sources on emulator H_2, computed in time $O(|S| \cdot n^{1+o(1)})$.

This lets us maintain $(1+\epsilon)$-MSSP from up to $O(n^{0.52})$ sources in almost (up to an $n^{o(1)}$ factor) the same running time as maintaining distances from a single-source by computing multi-source distances statically on this very sparse emulator and querying small distances from the algebraic data structure. This approach naturally extends to maintaining all-pairs distances deterministically and yields a worst-case update time of $n^{2+o(1)}$ by setting $S = V$.

Having described our approach for maintaining the more general emulator, let us briefly explain the differences to the dynamic spanner algorithm of [16]: We do not aim at *directly* maintaining an almost linear-size spanner. Instead, we use a two-level scheme in which we first compute a $(1+\epsilon, 4)$-emulator of "medium" sparsity (outlined in Section II-A) and then resparsify this first-level emulator with a static algorithm. Hence, we get improved bounds for the second-level (near-linear size) emulators, since we can maintain the "first-level" emulators more efficiently than the algorithms in [16] due to the properties described in Section II-A. Moreover, our deterministic dynamic hitting set and our algebraic data structure supporting its changes let us maintain these emulators deterministically, whereas the spanners of [16] are randomized.

Diameter Approximation: Our sparse emulators can also be used to maintain a (nearly) $(3/2+\epsilon)$-approximation of the diameter. Our algorithm is an adaptation of the dynamic algorithm by [20], which is in turn based on an algorithm by [36]. At a high-level, we need to query (approximate) multi-source distances from three sets of size at most $O(\sqrt{n})$. We show that our emulators can be used to maintain such approximate distances much more efficiently than the data structures of [20].

$(1+\epsilon)$-APSP Distance Oracles: Finally, we maintain a data structure with worst-case subquadratic update time that supports sublinear all-pairs $(1+\epsilon)$-approximate distance queries. Our algorithm is based on ideas of [20], [54] that utilize well-known path hitting techniques (e.g. [44]). In order to get improved bounds we again use our sparse $(1+\epsilon, \beta)$-emulators. We need to handle some technicalities both in the algorithm and its analysis introduced by the additive factor β, combined with h-bounded distances maintained in the algorithm of [20] for an appropriately chosen parameter h.

III. Approximate Distances via Emulators

In this section we focus on maintaining emulators with various tradeoffs and describing how they can be combined with the algebraic data structure of Lemma 2.2 for obtaining dynamic st and single-source distance approximations.

While our main focus is on st-distances, as a warm-up we start with our SSSP result.

We first assume that we have a low-recourse dynamic hitting set which we use in maintaining $(1+\epsilon, 2)$-emulators (with application in $(1+\epsilon)$-SSSP) and $(1+\epsilon, 4)$-emulators (with applications in $(1+\epsilon)$-st). We will then move on to give a deterministic algorithm that maintains low-recourse hitting sets.

A. Deterministic $(1+\epsilon, 2)$-Emulators and $(1+\epsilon)$-SSSP

In this section we summarize how to maintain $(1+\epsilon)$-SSSP with a worst-case update time matching the conditional lower bound of [12]. We start by describing how to maintain a $(1+\epsilon, 2)$-emulator, assuming that we have a low-recourse hitting set, and can compute bounded-hop distances from elements in this set. The algorithm is summarized in Algorithm 1. The omitted details, including the hitting set algorithm, can be found in the full version. Assume that we are given two functions:

- UpdateHittingSet(G, d), which returns a dynamically maintained hitting set for neighborhoods of heavy nodes (i.e., with degree at least d) satisfying Lemma 2.1.

- QUERYDISTANCES(G, S, T, h), which can query h-bounded distances between pairs in $S \times T$ as specified in Lemma 2.2.

Algorithm 1: Update Algorithm for a $(1+\epsilon, 2)$-Emulator

Input : Unweighted Graph $G = (V, E)$;
$A_d :=$ UPDATEHITTINGSET(G, d) with $d = \sqrt{n \log n}$;
For all nodes $\{v : \deg(v) \leq d\}$, add all the edges incident to v to H with weight 1;
QUERYDISTANCES$(G, A_d, V, \lceil \frac{2}{\epsilon} \rceil + 1)$;
For all nodes $\{v : \deg(v) \geq d\}$, add the edge from v to the neighbor in A_d with weight 1;
QUERYDISTANCES$(G, A_d, V, \lceil \frac{2}{\epsilon} \rceil + 1)$;
Add edges $\{(u, w) : u \in A_d, w \in V, d_G(u, v) \leq \lceil \frac{2}{\epsilon} \rceil + 1\}$ to H, and set the weight of each edge (u, w) to $d_G(u, w)$;
return H;

Observe that even though we start with an unweighted graph, we need to add weighted edges to the emulator (with weight corresponding to the distance between the endpoints). We note that a similar, but randomized version of this emulator construction (working only against an oblivious adversary) was used in [32] for maintaining approximate shortest paths decrementally. We have,

Theorem 3.1: Given an unweighted graph $G = (V, E)$, $0 < \epsilon < 1$, we can deterministically maintain a $(1+\epsilon, 2)$-emulator with size $O(n^{3/2}\sqrt{\log n})$. The worst-case update time is $O((n^{\omega(1,1,\nu)-\nu} + n^{1+\nu})\epsilon^{-2} \log \epsilon^{-1})$ for any $0 \leq \nu \leq 1$ and preprocessing time is $O(n^{\omega}\epsilon^{-2} \log \epsilon^{-1})$.
For current bounds on ω and best choice of $\nu \approx 0.529$, this is $O(n^{1.529}\epsilon^{-2} \log \epsilon^{-1})$ update time. We can now use this result to prove Theorem 1.2, details of which can be found in the full version.

Using an emulator for maintaining $(1+\epsilon)$-SSSP.: Given an unweighted graph $G = (V, E)$, we first maintain a $(1 + \frac{\epsilon}{2}, 2)$-emulator H for G. Given a single-source s and H, we can now maintain the distances by:

(i) Using the algebraic data structure of Lemma 2.2: $\lceil 4/\epsilon \rceil$-hop bounded distances from s on G,
(ii) After each update, *statically* computing SSSP on H in $O(n^{3/2}\sqrt{\log n})$ time.

B. Deterministic $(1+\epsilon, 4)$-Emulator and $(1+\epsilon)$-st Distances

In this section, we give another emulator-based algorithm that lets us maintain the approximate distance from a given source s to a given destination t with better update time than the time bound we showed for SSSP. We maintain a sparser emulator with a slightly larger additive stretch that supports faster computation of the st distance approximation. In particular, we maintain a $(1+\epsilon, 4)$-emulator of size $\tilde{O}(n^{4/3})$.

Compared to the emulator of Section III-A, for this emulator construction we need to maintain bounded distances with our algebraic data structure for a smaller number of pairs of nodes, which increases efficiency. This, combined with the fact that our emulators are sparser, leads to a faster algorithm for maintaining $(1+\epsilon)$-approximate st distances.

$(1+\epsilon, 4)$-emulator.: We start by maintaining a sparse emulator with slightly larger additive stretch term. The algorithm is summarized in Algorithm 6.

Algorithm 2: Update Algorithm for $(1+\epsilon, 4)$-Emulators

Input : Unweighted Graph $G = (V, E)$.
$A_d :=$ UPDATEHITTINGSET(G, d) with $d = n^{1/3}\sqrt{\log n}$;
For all nodes $\{v : \deg(v) \geq d\}$, add the edge from v to a neighbor in A_d with weight 1;
For all nodes $\{v : \deg(v) \leq d\}$, add all the edges incident on v to H with weight 1;
QUERYDISTANCES$(G, A_d, A_d, \lceil \frac{4}{\epsilon} \rceil + 2)$;
Add edges $\{(u, w) : u, w \in A_d, d_G(u, v) \leq \lceil \frac{4}{\epsilon} \rceil + 2\}$ to H, and set the weight of each edge (u, w) to $d_G(u, w)$;
return H;

Assuming that we can maintain a low-recourse hitting set A_d and $O(1/\epsilon)$-bounded distance between pairs $A_d \times A_d$, Algorithm 6 leads to an emulator with the following guarantees (proof can be found in the full version):

Theorem 3.2: Given an unweighted graph $G = (V, E)$, $0 < \epsilon < 1$, we can deterministically maintain a $(1+\epsilon, 4)$-emulator of size $O(n^{4/3}\sqrt{\log n})$ with worst-case update time of $O((n^{\omega(1,1,\mu)-\mu} + n^{\omega(1,\mu,\nu)-\nu} + n^{\mu+\nu} + n^{4/3})\epsilon^{-2} \log \epsilon^{-1})$ for any $0 \leq \nu \leq \mu \leq 1$, and preprocessing time of $O(n^{\omega}\epsilon^{-2} \log \epsilon^{-1})$.
For current bounds on ω and $\mu \approx 0.856$, $\nu \approx 0.551$, this is $O(n^{1.407}\epsilon^{-2} \log \epsilon^{-1})$ update time. The proof of Theorem 1.1 follows from a similar approach discussed for $(1+\epsilon)$-SSSP by taking the minimum of two estimates.

IV. SPARSE EMULATOR WITH APPLICATIONS IN $(1+\epsilon)$-APSP AND $(1+\epsilon)$-MSSP

In this section we show that by maintaining a much sparser emulator, we can maintain distances from many sources efficiently. At a high-level, we first use the construction in the previous section to maintain a $(1+\epsilon, 4)$-emulator H, and then use a static deterministic algorithm on H to obtain a $(1+\epsilon, n^{o(1)})$-emulator with size $n^{1+o(1)}$.

A. Sparse Deterministic Emulators

We start by showing that we can maintain near-additive emulators with general stretch/size tradeoffs. Before describing our dynamic construction, we observe that statically we can construct near-additive spanners (and hence emulators) efficiently and deterministically. For this we can use the deterministic algorithm of [53] for constructing the clusters used in the *spanner construction* of [22]. In other words, we can derandomize the spanner construction in [22] and have:

Lemma 4.1 ([22], [53]): Given an unweighted graph $G = (V, E)$ with m edges, and an integer $k > 1$, there is a deterministic algorithm that constructs, for any $0 < \epsilon \leq 1$, a $(1+\epsilon, \beta)$-spanner with $\beta = (1/\epsilon)^k$ and size $O(n^{1+1/k})$ in $\tilde{O}(mn^{1/k}))$ time.

Note that while with this (static) lemma we can construct a spanner (i.e., is a true subgraph of G), in our deterministic dynamic construction the algebraic techniques only let us maintain an emulator efficiently. For maintaining these emulators we perform the following for a parameter ϵ' that will be set later:

- Maintain a $(1+\epsilon', 4)$-emulator H of size $\tilde{O}(n^{4/3})$.
- Turn the emulator into an unweighted graph: we replace each *weighted edge* $w(e)$ of H with an unweighted path of length $w(e)$. Since the emulator we constructed only has edge weights bounded by $O(1/\epsilon')$, this will blow-up the size only by a factor of $O(1/\epsilon')$.
- Using Lemma 4.1, we statically construct a $(1+\epsilon', \beta)$-emulator H' of H.

It is easy to see that H' is now an emulator of G with slightly larger additive factor. More formally,

Lemma 1.4: Given an unweighted, undirected graph $G = (V, E)$, parameters $0 < \epsilon < 1$ and $2 \leq k \leq \log n$, we can maintain a $(1+\epsilon, \beta)$-emulator of G with size $\tilde{O}(n^{1+1/k})$, where $\beta = O(1/\epsilon)^k$ deterministically with worst-case update time of $\max(\tilde{O}(n^{4/3+1/k}), O(n^{1.407}\epsilon^{-2}\log\epsilon^{-1}))$. Here the latter term of the update time has the same dependence on ω as Theorem 1.1. The preprocessing time of this algorithm is $O(n^{\omega}\epsilon^{-2}\log\epsilon^{-1})$.

Proof: It is easy to see that the update time is the maximum of the time required for running the algorithm in Lemma 4.1 statically, and the dynamic time for maintaining the $(1+\epsilon, 4)$-emulator which is given by Theorem 3.2.

For every pair of nodes s, t, there is a path with length $(1+\epsilon')d_H(s,t) + \beta$ in H'. We also know $d_H(s,t) \leq (1+\epsilon')d_G(s,t) + 2$. Hence,

$$\begin{aligned} d'_H(s,t) &\leq (1+\epsilon')d_H(s,t) + \beta \\ &\leq (1+\epsilon')[(1+\epsilon')d_G(s,t) + 2] + \beta \\ &\leq (1+3\epsilon')d_G(s,t) + O(\beta) \end{aligned}$$

Hence by setting $\epsilon' = \epsilon/3$ the claim follows. ∎

One important special case of this result is when we set $k = \sqrt{\log n}$ and ϵ is a constant. In this case we have an additive stretch of $\beta = O(1/\epsilon)^{\sqrt{\log n}} = n^{o(1)}$, and the size of the emulator is $\tilde{O}(n^{1+o(1)})$. We can obtain such a sparse emulator in $O(\frac{n^{1.407}}{\epsilon^2 \log \epsilon^{-1}})$ worst-case update time deterministically.

B. Deterministic $(1+\epsilon)$-MSSP

Now using the emulator in this special setting, we do the following for maintaining multi-source distances from a set S of sources:

1) At each update, after updating H', statically compute $S \times V$ distances on H' in $O(|S| \cdot |E(H')|)$ time.
2) Maintain $O(\beta)$-bounded distances between pairs in $S \times V$ on G.
3) The distance estimate $d(s, v)$ for each source $s \in S$ and node v is the minimum distance estimate derived from these two steps.

It is now easy to combine Lemma 1.4 with Lemma 2.2 by setting $k = \sqrt{\frac{\log_{1/\epsilon} n}{2}}$ and thus $\beta = O(1/\epsilon)^{\sqrt{2\log_{1/\epsilon} n}}$ to prove Theorem 1.5. Hence, we can compute $(1+\epsilon)$-MSSP from up to $O(n^{0.52})$ sources in the same time complexity as our $(1+\epsilon)$-SSSP algorithm, namely $O(n^{1.529})$.

Deterministic $(1+\epsilon)$-APSP: We can directly use our $(1+\epsilon)$-MSSP algorithm to maintain all-pairs-shortest paths distances deterministically by setting $S = V$.

REFERENCES

[1] J. v. d. Brand, S. Forster, and Y. Nazari, "Fast deterministic fully dynamic distance approximation," *CoRR*, vol. abs/2111.03361, 2021, https://arxiv.org/abs/2111.03361.

[2] S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson, "On the power of randomization in on-line algorithms," *Algorithmica*, vol. 11, no. 1, pp. 2–14, 1994, announced at STOC 1990.

[3] S. Bhattacharya, M. Henzinger, and D. Nanongkai, "Fully dynamic approximate maximum matching and minimum vertex cover in $o(\log^3 n)$ worst case update time," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2017, pp. 470–489.

[4] S. Bhattacharya, D. Chakrabarty, M. Henzinger, and D. Nanongkai, "Dynamic algorithms for graph coloring," in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2018, pp. 1–20.

[5] S. Sawlani and J. Wang, "Near-optimal fully dynamic densest subgraph," in *Proccedings of the 52nd Symposium on Theory of Computing (STOC)*, 2020, pp. 181–193.

[6] J. Chuzhoy, Y. Gao, J. Li, D. Nanongkai, R. Peng, and T. Saranurak, "A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond," in *Proceedings of the 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS 2020)*, 2020, pp. 1158–1167.

[7] W. Jin and X. Sun, "Fully dynamic s-t edge connectivity in subpolynomial time," in *Proceedings of the 62nd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2021.

[8] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd," in *STOC*. ACM, 2012, pp. 887–898.

[9] F. L. Gall, "Powers of tensors and fast matrix multiplication," in *ISSAC*. ACM, 2014, pp. 296–303.

[10] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication," in *SODA*. SIAM, 2021, pp. 522–539.

[11] F. L. Gall and F. Urrutia, "Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor," in *SODA*. SIAM, 2018, pp. 1029–1046.

[12] J. v. d. Brand, D. Nanongkai, and T. Saranurak, "Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds," in *FOCS*. IEEE Computer Society, 2019, pp. 456–480.

[13] P. Sankowski, "Dynamic transitive closure via dynamic matrix inverse (extended abstract)," in *FOCS*. IEEE Computer Society, 2004, pp. 509–517.

[14] ——, "Subquadratic algorithm for dynamic shortest distances," in *COCOON*, ser. Lecture Notes in Computer Science, vol. 3595. Springer, 2005, pp. 461–470.

[15] ——, "Faster dynamic matchings and vertex connectivity," in *SODA*. SIAM, 2007, pp. 118–126.

[16] T. Bergamaschi, M. Henzinger, M. P. Gutenberg, V. V. Williams, and N. Wein, "New techniques and fine-grained hardness for dynamic near-additive spanners," in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2021, pp. 1836–1855.

[17] R. Seidel, "On the all-pairs-shortest-path problem in unweighted undirected graphs," *Journal of Computer and System Sciences*, vol. 51, no. 3, pp. 400–403, 1995, announced at STOC 1992.

[18] A. Karczmarz, A. Mukherjee, and P. Sankowski, "Subquadratic dynamic path reporting in directed graphs against an adaptive adversary," in *STOC*. ACM, 2022.

[19] P. Sankowski, "Algebraic graph algorithms," in *MFCS*, ser. Lecture Notes in Computer Science, vol. 5162. Springer, 2008, pp. 68–82.

[20] J. v. d. Brand and D. Nanongkai, "Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time," in *FOCS*. IEEE Computer Society, 2019, pp. 436–455.

[21] M. Elkin and D. Peleg, "$(1+\epsilon, \beta)$-spanner constructions for general graphs," *SIAM Journal on Comput.*, vol. 33, no. 3, pp. 608–631, 2004, announced at STOC 2001.

[22] M. Thorup and U. Zwick, "Spanners and emulators with sublinear distance errors," in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, 2006, p. 802–809.

[23] M. Elkin and O. Neiman, "Efficient algorithms for constructing very sparse spanners and emulators," *ACM Transactions on Algorithms (TALG)*, vol. 15, no. 1, pp. 1–29, 2018.

[24] G. Ausiello, P. G. Franciosa, and G. F. Italiano, "Small stretch spanners on dynamic graphs," *Journal of Graph Algorithms and Applications*, vol. 10, no. 2, pp. 365–385, 2006, announced at ESA 2005.

[25] M. Elkin, "Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners," *ACM Transactions on Algorithms*, vol. 7, no. 2, pp. 20:1–20:17, 2011, announced at ICALP 2007.

[26] S. Baswana, S. Khurana, and S. Sarkar, "Fully dynamic randomized algorithms for graph spanners," *ACM Transactions on Algorithms*, vol. 8, no. 4, pp. 35:1–35:51, 2012.

[27] G. Bodwin and S. Krinninger, "Fully dynamic spanners with worst-case update time," in *Proc. of the 24th European Symposium on Algorithms (ESA 2016)*, vol. 57, 2016.

[28] A. Bernstein, S. Forster, and M. Henzinger, "A deamortization approach for dynamic spanner and dynamic maximal matching," *ACM Transactions on Algorithms*, vol. 17, no. 4, pp. 29:1–29:51, 2021, announced at SODA 2019.

[29] S. Forster and G. Goranci, "Dynamic low-stretch trees via dynamic low-diameter decompositions," in *Proc. of the 51st Annual ACM SIGACT Symposium on Theory (STOC 2019)*, 2019, pp. 377–388.

[30] A. Bernstein, J. van den Brand, M. P. Gutenberg, D. Nanongkai, T. Saranurak, A. Sidford, and H. Sun, "Fully-dynamic graph sparsifiers against an adaptive adversary," *CoRR*, vol. abs/2004.08432, 2020.

[31] A. Bernstein and L. Roditty, "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)*, 2011, pp. 1355–1365.

[32] M. Henzinger, S. Krinninger, and D. Nanongkai, "Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization," *SIAM Journal on Computing*, vol. 45, no. 3, pp. 947–1006, 2016, announced at FOCS 2013.

[33] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," *Journal of the ACM*, vol. 51, no. 6, pp. 968–992, 2004, announced at STOC 2003.

[34] M. Thorup, "Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles," in *Proceedings of the 9th Scandinavian Workshop on Algorithm (SWAT 2004)*, 2004, pp. 384–396.

[35] M. P. Gutenberg and C. Wulff-Nilsen, "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)*, 2020, pp. 2562–2574.

[36] L. Roditty and V. Vassilevska Williams, "Fast approximation algorithms for the diameter and radius of sparse graphs," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 515–524.

[37] B. Ancona, M. Henzinger, L. Roditty, V. V. Williams, and N. Wein, "Algorithms and hardness for diameter in dynamic graphs," in *ICALP*, ser. LIPIcs, vol. 132. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 13:1–13:14.

[38] K. Choudhary and O. Gold, "Extremal distances in directed graphs: Tight spanners and near-optimal approximation algorithms," in *SODA*. SIAM, 2020, pp. 495–514.

[39] A. Abboud and V. V. Williams, "Popular conjectures imply strong lower bounds for dynamic problems," in *Proc. of the 55th Symposium on Foundations of Computer Science (FOCS)*, 2014.

[40] S. Bhattacharya, M. Henzinger, D. Nanongkai, and X. Wu, "Dynamic set cover: Improved amortized and worst-case update time," in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2021, pp. 2537–2549.

[41] V. King and G. Sagert, "A fully dynamic algorithm for maintaining the transitive closure," *Journal of Computer and System Sciences*, vol. 65, pp. 150–167, 2002, announced at STOC 1999.

[42] C. Demetrescu and G. F. Italiano, "Fully dynamic transitive closure: Breaking through the $o(n^2)$ barrier," in *FOCS*. IEEE Computer Society, 2000, pp. 381–389.

[43] D. Dor, S. Halperin, and U. Zwick, "All-pairs almost shortest paths," *SIAM Journal on Computing*, vol. 29, no. 5, pp. 1740–1759, 2000, announced at FOCS 1996.

[44] J. D. Ullman and M. Yannakakis, "High-probability parallel transitive-closure algorithms," *SIAM Journal on Computing*, vol. 20, no. 1, pp. 100–125, 1991, announced at SPAA 1990.

[45] A. Abboud, R. Addanki, F. Grandoni, D. Panigrahi, and B. Saha, "Dynamic set cover: improved algorithms and lower bounds," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, 2019, pp. 114–125.

[46] S. Bhattacharya, M. Henzinger, and D. Nanongkai, "A new deterministic algorithm for dynamic set cover," in *60th Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2019, pp. 406–423.

[47] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi, "Online and dynamic algorithms for set cover," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, 2017, pp. 537–550.

[48] J. v. d. Brand and T. Saranurak, "Sensitive distance and reachability oracles for large batch updates," in *FOCS*. IEEE Computer Society, 2019, pp. 424–435.

[49] Y. Gu and H. Ren, "Constructing a distance sensitivity oracle in $o(n^{2.5794}m)$ time," in *ICALP*, ser. LIPIcs, 2021.

[50] D. E. Knuth, *The art of computer programming. Vol.2. Seminumerical algorithms*. Addison-Wesley Professional, 1997.

[51] J. Sherman and W. J. Morrison, "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix," *The Annals of Mathematical Statistics*, vol. 21, no. 1, pp. 124–127, 1950.

[52] M. A. Woodbury, *Inverting modified matrices*. Statistical Research Group, 1950.

[53] L. Roditty, M. Thorup, and U. Zwick, "Deterministic constructions of approximate distance oracles and spanners," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2005, pp. 261–272.

[54] L. Roditty and U. Zwick, "Dynamic approximate all-pairs shortest paths in undirected graphs," *SIAM Journal on Computing*, vol. 41, no. 3, pp. 670–683, 2012.