

# On Incremental Approximate Shortest Paths in Directed Graphs

Adam Grkiewicz\*

Adam Karczmarz<sup>†</sup>

## Abstract

In this paper, we show new data structures maintaining approximate shortest paths in sparse directed graphs with polynomially bounded non-negative edge weights under edge insertions.

We give more efficient incremental  $(1 + \epsilon)$ -approximate APSP data structures that work against an adaptive adversary: a deterministic one with  $\tilde{O}(m^{3/2}n^{3/4})$ <sup>1</sup> total update time and a randomized one with  $\tilde{O}(m^{4/3}n^{5/6})$  total update time. For sparse graphs, these both improve polynomially upon the best-known bound against an adaptive adversary [?]. To achieve that, building on the ideas of [?, ?], we show a near-optimal  $(1 + \epsilon)$ -approximate incremental SSSP data structure for a special case when all edge updates are adjacent to the source, that might be of independent interest.

We also describe a very simple and near-optimal offline incremental  $(1 + \epsilon)$ -approximate SSSP data structure. While online near-linear partially dynamic SSSP data structures have been elusive so far (except for dense instances), our result excludes using certain types of impossibility arguments to rule them out. Additionally, our offline solution leads to near-optimal and deterministic all-pairs bounded-leg shortest paths data structure for sparse graphs.

## 1 Introduction

Computing shortest paths is one of the most classical algorithmic problems on graphs, with numerous applications in both theoretical and practical scenarios. Dynamic variants of the shortest paths problem are undoubtedly among the most extensively studied dynamic graph problems.

In dynamic shortest paths problems, the goal is to design a data structure that maintains (or provides efficient query access to) the desired information about the shortest paths in a graph  $G = (V, E)$  subject to edge set updates. If both edge insertions and deletions on  $G$  are supported, the data structure is called fully dynamic. If only edge insertions or only edge deletions are allowed, it is called incremental or decremental, respectively. Incremental and decremental settings are together referred to as partially dynamic. In the fully dynamic setting, one seeks data structures with low (amortized) update time, whereas in partially dynamic settings, the typical goal is to optimize total update time, i.e., the time needed to process an entire sequence of updates.

While we would ideally like to have efficient fully dynamic data structures, their existence<sup>2</sup> is often ruled out by conditional lower bounds (e.g., [?, ?, ?]). Partially dynamic scenarios are often

---

\*University of Wrocław, Poland. Work partially done when the author was a student scholarship recipient at IDEAS NCBR supported by the National Science Centre (NCN) grant no. 2022/47/D/ST6/02184.

<sup>†</sup>University of Warsaw and IDEAS NCBR, Poland. Supported by the National Science Centre (NCN) grant no. 2022/47/D/ST6/02184.

<sup>1</sup>Throughout, for convenience we use the standard notation  $\tilde{O}(Y)$  as a shorthand for the expression  $O(Y \text{ polylog } n)$ .

<sup>2</sup>Or their feasibility without resorting to complex and impractical tools such as fast matrix multiplication [?].

more approachable algorithmically, while still being very useful in applications (such as max-flow, e.g., [?, ?]), including designing the said fully dynamic data structures.

When analyzing randomized dynamic algorithms, one needs to specify the assumptions about how an adversary may adapt to the data structure’s outputs for the data structure to work correctly and efficiently. An adaptive adversary can adjust the sequence of updates and queries it issues freely based on the preceding data structure’s outputs. An oblivious adversary has to fix the entire update sequence before the process starts (but without revealing it upfront). Adaptive data structures are more desirable; e.g., they can be applied in a black-box manner when designing static algorithms.

## 1.1 Incremental shortest paths

In this paper, we focus on incremental shortest paths data structures for weighted directed graphs.

**Exact data structures.** Even in incremental weighted digraphs, maintaining shortest paths exactly is computationally challenging. As proved recently by [?], one cannot even exactly maintain the distance between a single fixed source-target pair in an incremental graph within  $O(m^{2-\epsilon})$  total update time (for any density  $m$ ), i.e., significantly faster than recomputing from scratch. The lower bound is conditional on the Min-Weight 4-Clique Hypothesis (eg. [?, ?, ?]) and holds even in the offline setting, that is, when the update sequence is revealed upfront. A folklore observation is that the all-pairs shortest paths (APSP) matrix can be updated after an edge insertion in  $O(n^2)$  time, and hence this yields an incremental APSP data structure with  $O(mn^2)$  total update time. This is optimal if the  $n^2$  distances are to be maintained explicitly.

Non-trivial exact data structures are known for the special case of unweighted digraphs. Single-source shortest paths (SSSP) in unweighted digraphs can be maintained in  $O(nm)$  total time [?, ?], whereas APSP can be maintained in  $\tilde{O}(n^3)$  total time [?]. Both these bounds are near-optimal if the distances are to be maintained explicitly. There is also an  $n^{2-o(1)}$  lower bound for maintaining a single source-target distance in an incremental sparse digraph [?].

**Approximate SSSP.** The lack of non-trivial approaches for exactly maintaining shortest paths in incremental weighted digraphs motivates studying approximate data structures. For SSSP, the classical ES-tree [?, ?] can be generalized to partially dynamic weighted digraphs with real weights in  $[1, W]$  at the cost of  $(1 + \epsilon)$ -approximation (where  $\epsilon = O(1)$ ) and has  $\tilde{O}(mn \log(W))$  total update time (see, e.g., [?, ?]). While quadratic in the general case, the approximate ES-tree is rather flexible and might be set up to run much faster – in  $\tilde{O}(mh \log(W))$  total time – if guaranteed that shortest (or short enough) paths from  $s$  use at most  $h$  hops.

Henzinger, Krinninger and Nanongkai [?, ?] were the first to break through the quadratic bound of the ES-tree polynomially, also in the decremental setting, albeit only against an oblivious adversary. Probst Gutenberg, Vassilevska Williams, and Wein [?] showed the first SSSP data structure designed specifically for the incremental setting against an adaptive adversary, with  $\tilde{O}(n^2 \log(W))$  total update time. Their data structure is deterministic and near-optimal for dense graphs. Chechik and Zhang [?] showed two data structures for sparse graphs: one deterministic with  $\tilde{O}(m^{5/3} \log(W))$  total update time and another Monte Carlo randomized with  $\tilde{O}((mn^{1/2} + m^{7/5}) \log(W))$  total update time, correct w.h.p.<sup>3</sup> against an adaptive adversary. These two bounds were subsequently improved to  $\tilde{O}(m^{3/2} \log(W))$  (deterministic) and  $\tilde{O}(m^{4/3} \log(W))$  (randomized) respectively by Kyng, Meierhans and Probst Gutenberg [?].

---

<sup>3</sup>With high probability, that is, with probability  $1 - n^{-c}$  for any desired constant  $c \geq 1$ .

Very recently, [?] gave a deterministic  $(1 + \epsilon)$ -approximate incremental min-cost flow algorithm implying that a  $(1 + \epsilon)$ -approximate shortest path for a single source-target pair in polynomially weighted digraphs can be maintained in almost optimal  $m^{1+o(1)}$  time.

No non-trivial conditional lower bounds for incremental approximate SSSP have been described and thus the existence of an SSSP data structure with near-linear total update time remains open.

**Approximate APSP.** Bernstein [?] showed a Monte Carlo randomized data structure maintaining  $(1 + \epsilon)$ -approximate APSP in either incremental or decremental setting with  $\tilde{O}(mn \log(W))$  total update time. This bound is near-optimal (even for incremental transitive closure [?]), but the data structure gives correct answers w.h.p. only against an oblivious adversary. A deterministic  $(1 + \epsilon)$ -approximate data structure with  $\tilde{O}(n^3 \log(W))$  total update time is known [?]. This is near-optimal for dense graphs [?]. [?] showed a deterministic data structure with  $\tilde{O}(mn^{4/3} \log^2(W))$  total update time that is more efficient for sparse graphs.

Interestingly, the incremental APSP data structures for sparse digraphs [?, ?] are both obtained without resorting to any of the critical ideas from the state-of-the-art sparse incremental SSSP data structures [?]. Instead, they work by running approximate ES-trees on carefully selected instances where approximately shortest paths from the source have sublinear numbers of hops. Note that a deterministic bound  $\tilde{O}(nm^{3/2} \log(W))$  and a randomized adaptive bound  $\tilde{O}(nm^{4/3} \log(W))$  could be achieved by simply employing the two data structures of [?] for all possible sources  $s \in V$ . This black-box application does not improve upon [?], though. It is thus interesting to ask whether the data structures of [?, ?], just like the ES-tree, could be adjusted to run faster in some special cases that are useful in designing all-pairs data structures.

## 1.2 Our contribution

In this paper, we give new incremental  $(1 + \epsilon)$ -approximate shortest paths data structures for sparse directed graphs. With the goal of obtaining improved all-pairs data structures, we first identify a useful special case of the incremental SSSP problem for which a data structure with near-linear total update time is achievable. Building on the ideas from [?, ?], we prove:

**Theorem 1.1** (Simplified version of Theorem ??). Let  $\epsilon \in (0, 1)$ . Let  $G = (V, E)$  be a digraph with edge weights in  $\{0\} \cup [1, W]$  and a source  $s \in V$ . There exists a deterministic data structure explicitly maintaining distance estimates  $d : V \rightarrow \mathbb{R}_{\geq 0}$  satisfying

$$\text{dist}_G(s, v) \leq d(v) \leq (1 + \epsilon) \cdot \text{dist}_G(s, v)$$

for all  $v \in V$  and only supporting insertions (or weight decreases) of source edges  $e = sv$ ,  $v \in V$ .

The total update time of the data structure is  $O(m \log(nW) \log^2(n)/\epsilon + \Delta)$ , where  $m$  is the final number of edges in  $G$  and  $\Delta$  is the total number of updates issued.

Even though the repertoire of possible updates in ?? seems very limited, we remark that in the exact setting, explicitly maintaining distances under source updates like this requires  $\Omega(mn)$  total time since one could reduce static APSP to a sequence of  $O(n)$  updates of this kind.<sup>4</sup>

We also show (??) that without compromising the total update time, ?? can be extended – at the cost of slight accuracy decline – to also accept arbitrary edge insertions, as long as they do not improve the distances from the source too much. With this observation, we can combine ?? with the near-optimal APSP data structure for dense digraphs [?] and obtain improved incremental APSP data structure for sparse graphs against an adaptive adversary:

<sup>4</sup>This is precisely the argument used in [?] to reduce static APSP to partially dynamic SSSP.

Theorem 1.2. Let  $G = (V, E)$  be a digraph with edge weights in  $\{0\} \cup [1, W]$  and let  $\epsilon \in (0, 1)$ . There exist data structures explicitly maintaining distance estimates  $d : V \times V \rightarrow \mathbb{R}_{\geq 0}$  such that

$$\text{dist}_G(u, v) \leq d(u, v) \leq (1 + \epsilon) \cdot \text{dist}_G(u, v)$$

for all  $u, v \in V$ , subject to edge insertions or weight decreases issued to  $G$ :

- A deterministic one with  $O(m^{3/2}n^{3/4} \log^2(nW) \log(n)/\epsilon^{3/2} + \Delta)$  total update time.
- A Monte Carlo randomized one that produces correct answers against an adaptive adversary (whp.) with  $O(m^{4/3}n^{5/6} \log^3(nW) \log(n)/\epsilon^{7/3} + \Delta)$  total update time.

Here,  $m$  denotes the final number of edges in  $G$  and  $\Delta$  the total number of updates issued.

For sparse graphs, the deterministic variant improves upon the total update time of the data structure of [?] by a factor of  $n^{1/12}$ . By employing the randomized technique of preventing error buildup due to [?], the speed-up over [?] is increased to a factor of  $n^{1/6}$ .

We believe that ?? and its extensions may find other applications in the future.

Offline data structures. Finally, we also prove that the  $(1 + \epsilon)$ -approximate incremental SSSP problem has a near-linear deterministic offline solution. Formally, we show:

Theorem 1.3. Let  $\epsilon \in (0, 1)$ . Let  $G = (V, E)$  be a digraph with edge weights in  $\{0\} \cup [1, W]$  and a source  $s \in V$ . Suppose we are given a sequence of  $\Delta$  incremental updates to  $G$  (edge insertions or weight decreases). Let  $G^0, \dots, G^\Delta$  denote the subsequent versions of the graph  $G$ .

Then in  $O(m \log(n) \log(nW) \log^2(\Delta)/\epsilon + \Delta)$  deterministic time one can compute a data structure supporting queries  $(v, j)$  about a  $(1 + \epsilon)$ -approximate estimate of  $\text{dist}_{G^j}(s, v)$ . The query time is  $O(\log(\log(nW) \log(\Delta)/\epsilon))$ . For example, if  $\epsilon, W, \Delta \in \text{poly}(n)$ , the query time is  $O(\log \log n)$ .

Note that incremental and decremental settings are equivalent in the offline scenario, so the above works for decremental update sequences as well. To the best of our knowledge, no prior<sup>5</sup> work described offline partially dynamic  $(1 + \epsilon)$ -approximate SSSP algorithms. It is worth noting that for dense graphs, the best known online data structures [?, ?] already run in near-optimal  $\tilde{O}(n^2 \log(W))$  time. However, the state-of-the-art partially dynamic SSSP bounds [?, ?, ?] are still off from near-linear by polynomial factors.

?? implies that if incremental or decremental approximate SSSP happen to not have near-linear algorithms, then showing a (conditional) lower bound for the problem requires exploiting the online nature of the problem. For example, showing reductions from such problems as the Min-Weight  $k$ -Clique or  $k$ -cycle detection (as used in [?] and [?], respectively, for exact shortest path) cannot yield non-trivial lower bounds for partially dynamic SSSP.

Offline incremental APSP and its special cases have been studied before under the name all-pairs shortest paths for all flows (APSP-AF) [?, ?], or – in a special case when edges are inserted sorted by their weights – all-pairs bounded-leg shortest paths (APBLSP) [?, ?, ?]. For polynomially weighted digraphs, all known non-trivial data structures are  $(1 + \epsilon)$ -approximate. Duan and Ren [?] described a deterministic data structure with  $\tilde{O}(n^{(\omega+3)/2} \log(W))$  preprocessing and  $O(\log \log(nW))$  query time.<sup>6</sup> For sparser graphs with polynomial weights, the randomized partially dynamic data structure of [?] (augmented with persistence [?] to enable answering queries about individual graph’s versions) has  $\tilde{O}(mn)$  total update time and remains the state of the art, also for APBLSP. By applying ?? for each source, one obtains an APSP-AF data structure that is much simpler, deterministic, and a log-factor faster than [?].

<sup>5</sup>An independent and parallel work [?] established a similar result on offline incremental SSSP.

<sup>6</sup>Here,  $\omega$  denotes the matrix multiplication exponent. Currently, it is known that  $\omega \leq 2.372$  [?].

Finally, let us remark that even though, as presented in this paper, all the data structures we develop only maintain or output distance estimates, they can be easily extended to enable reporting a requested approximately shortest path  $P$  in near-optimal  $\tilde{O}(|P|)$  time.

### 1.3 Further related work

There is a broad literature on dynamic shortest paths data structures for weighted digraphs in fully dynamic (e.g., [?, ?, ?, ?, ?, ?]) and decremental (e.g., [?, ?, ?, ?]) settings. Much of the previous (and in particular very recent) work has been devoted to shortest paths data structures for undirected graphs, often with larger approximation factors, e.g., [?, ?, ?, ?, ?, ?, ?, ?].

### 1.4 Organization

In ?? we set some notation. Sections ?? and ?? together contain the description of the data structure for source-incident insertions and its extensions. In ?? we give our incremental APSP data structure. Finally, ?? describes the offline incremental SSSP data structure.

## 2 Preliminaries

In this paper, we only deal with non-negatively weighted digraphs  $G = (V, E)$ . We write  $e = uv$  to denote a directed edge from its tail  $u$  to its head  $v$ . We denote by  $w(e)$  the weight of edge  $e$ . By  $G + F$  we denote the graph  $G$  with edges  $F$  added. We also write  $G + e$  to denote  $G + \{e\}$ .

We define a path  $P = s \rightarrow t$  in  $G$  to be a sequence of edges  $e_1 \dots e_k$ , where  $u_i v_i = e_i \in E$ , such that  $v_i = u_{i+1}$ ,  $u_1 = s$ ,  $v_k = t$ . If  $s = t$ , then  $k = 0$  is allowed. Paths need not be simple, i.e., they may have vertices or edges repeated. We often view paths as subgraphs and write  $P \subseteq G$ .

The weight (or length)  $w(P)$  of a path  $P$  equals the sum  $\sum_{i=1}^k w(e_i)$ . We denote by  $\text{dist}_G(s, t)$  the weight of the shortest  $s \rightarrow t$  path in  $G$ . If the graph in question is clear from context, we sometimes omit the subscript and write  $\text{dist}(s, t)$  instead.

## 3 Dijkstra-like propagation

In this section we define and analyze properties of a Dijkstra-like procedure Propagate whose variant has been used by [?, ?] for obtaining state-of-the-art incremental SSSP data structures. We will use that in our specialized incremental SSSP data structures in Section ??.

Let  $G = (V, E)$  be a weighted directed graph with edge weights in  $\{0\} \cup [1, W]$ . Let  $\xi \in (0, 1)$  be an accuracy parameter, and let us put  $\ell := \lceil \log_{1+\xi}(nW) \rceil$ .

Let  $d : V \rightarrow \{0\} \cup [1, nW]$  be some estimate vector. We would like the estimates to satisfy and maintain – subject to edge insertions (or weight decreases) issued to  $G$  – the following invariant:

**Invariant 3.1.** For every edge  $uv = e \in E$ ,  $d(v) \leq (1 + \xi)(d(u) + w(e))$ .

The basic purpose of the procedure Propagate (see Algorithm ??) is to decrease some of the estimates  $d(\cdot)$  so that ?? (potentially broken by edge insertions) is again satisfied.

Roughly speaking, Propagate propagates estimate updates similarly to Dijkstra’s algorithm, but only decreases a given estimate if the new value is smaller than the old one by a factor at least  $1 + \xi$  or if the vertex it corresponds to is currently in the queue. For  $V_{\text{input}} \subseteq V$ ,  $\text{Propagate}(V_{\text{input}})$  initializes the queue with vertices from the set  $V_{\text{input}}$ . It returns a subset  $V_{\text{touched}} \subseteq V$  of vertices that have been explored (i.e., inserted to the queue) at any point during the procedure’s run.

Observation 3.2. Suppose an edge  $e = uv$  is inserted to  $G$ . If  $d(v) > (1 + \xi)(d(u) + w(e))$ , then setting  $d(v) := d(u) + w(e)$  followed by calling  $\text{Propagate}(\{v\})$  ensures ?? is satisfied.

Our data structures later on will rely on the following stronger property of  $\text{Propagate}$ .

Lemma 3.3. After running  $\text{Propagate}(V_{\text{input}})$ , for every edge  $uv = e \in E$  such that  $u, v \in V_{\text{touched}}$ ,  $e$  is relaxed, i.e., we have  $d(v) \leq d(u) + w(e)$ .

Proof. **TOPROVE 0** □

We call a path  $P = u \rightarrow v$  in  $G$  relaxed if  $d(v) \leq d(u) + w(P)$ . Note that if all edges of  $P$  are relaxed then by chaining inequalities for the individual edges we obtain that  $P$  is relaxed as well.

The following bounds the total time needed to maintain the estimates subject to edge insertions.

Lemma 3.4. If edge insertions issued to  $G$  are processed using ??, then the total time spent on maintaining the estimates  $d(\cdot)$  is  $O(n\ell \log n + m\ell) = O((m + n \log n) \log(nW)/\xi)$ .

Proof. **TOPROVE 1** □

## 4 An SSSP data structure for source insertions

In this section, inspired by the deterministic algorithm of [?], we describe an incremental SSSP data structure supporting only a very limited type of edge insertions, namely, insertions of edges originating directly in the source  $s$ . We first show a basic variant and then also describe some extensions that will be required in our incremental APSP data structure. We show:

Theorem 4.1. Let  $\xi \in (0, 1)$ . Let  $G = (V, E)$  be a digraph with weights in  $\{0\} \cup [1, W]$  and a source  $s \in V$ . There exists a data structure explicitly maintaining estimates  $d : V \rightarrow \mathbb{R}_{\geq 0}$  satisfying:

- $d(v)$  is the weight of some  $s \rightarrow v$  path in  $G$ , ie.,  $\text{dist}_G(s, v) \leq d(v)$ , and
- $d(v) \leq (1 + \xi)^{\log_2 m + 1} \text{dist}_G(s, v)$

for all  $v \in V$ . The data structure supports insertions (or weight decreases) of edges  $e = sv$ ,  $v \in V$ .

The total update time of the data structure is  $O(m \log(nW) \log n/\xi + \Delta)$ , where  $m$  is the final number of edges in  $G$  and  $\Delta$  is the total number of updates issued.

---

### Algorithm 1 $\text{Propagate}(V_{\text{input}})$

---

```

1: Initialize a priority queue  $Q = V_{\text{input}}$  keyed with  $d(\cdot)$ , and a set  $V_{\text{touched}} := V_{\text{input}}$ .
2: while  $Q \neq \emptyset$  do
3:    $u := Q.\text{pop}()$ 
4:   for  $uv = e \in E$  do
5:     if  $v \in Q$  then
6:        $d(v) := \min(d(v), d(u) + w(e))$  ▷ decrease key
7:     else if  $d(v) > (1 + \xi)(d(u) + w(e))$  then
8:        $d(v) := d(u) + w(e)$ 
9:   Insert  $v$  to  $Q$  and  $V_{\text{touched}}$ 
return  $V_{\text{touched}}$ 

```

---

---

**Algorithm 2** Source-Insert( $e = sv$ )

---

```

1:  $E := E \cup \{e\}$ 
2: if  $d(v) > (1 + \xi)w(e)$  then
3:    $d(v) := w(e)$ 
4:    $V_{\text{touched}} := \text{Propagate}(\{v\})$ 
5:   Increase the ranks of vertices  $V_{\text{touched}}$  to  $r + 1$ 
6:   Synchronize()

```

---

Remark 4.2. In ??, we bound the multiplicative error by  $(1 + \xi)^{\log_2 m + 1}$  merely for convenience:  $(1 + \xi)$  corresponds to the “multiplicative slack” on each edge maintained in ?. To achieve a target approximation factor of  $1 + \epsilon$  as in ??, it is enough to set  $\xi := \epsilon / (2 \log_2 m + 2)$ .<sup>7</sup> The total update time in terms of  $\epsilon$  then becomes  $O(m \log(nW) \log^2 n / \epsilon + \Delta)$ .

#### 4.1 Basic setup

For simplicity, let us assume that  $G$  initially contains  $n$  edges  $sv$  of weight  $nW$ , so that every vertex is reachable from  $s$  and  $\text{dist}(s, v) \leq nW$ . Note that unless  $v$  is not reachable from  $G$ , this assumption does not ever influence  $\text{dist}(s, v)$ . To drop the assumption, we might run a separate incremental single-source reachability data structure with linear total update time (such as [?]) and potentially return  $\infty$  instead of the maintained  $d(v)$  if  $v$  is not (yet) reachable from  $s$ .

The estimates  $d(v)$  the data structure maintains come from  $\{0\} \cup [1, nW]$ . They are initialized to  $d(v) := \text{dist}(s, v)$  using Dijkstra’s algorithm, so that  $d(s) = 0$  and  $d(v) \leq nW$ . The data structure’s operations will guarantee that the estimates never decrease and, moreover, each  $d(v)$  always corresponds to the length of some  $s \rightarrow v$  path in  $G$ . Consequently, we will have  $d(s) = 0$  and  $\text{dist}(s, v) \leq d(v)$  for  $v \in V$  at all times.

Let again  $\ell := \lceil \log_{1+\xi}(nW) \rceil$ . Crucially, the maintained estimates  $d(\cdot)$  also obey ??

#### 4.2 Handling source insertions

Consider the insertion pseudocode shown in ?. When an edge  $e = sv$  is inserted, the insertion is initially handled as explained in ?. This alone guarantees that ?? is satisfied and costs  $O(n\ell \log n + m\ell)$  time through all insertions (see ??). Afterwards, further steps (lines ?? and ?? in ??) are performed to reduce estimate errors accumulated on paths in  $G$ . Roughly speaking, these steps will interact with the estimates only via some additional Propagate calls. As a result, they cannot break ?? and thus we will assume it remains satisfied throughout. In the following, we explain the ideas behind these steps in detail.

Definition 4.3 (certificates). Let  $k \geq 0$  be an integer. We say that a vertex  $u \in V$  is  $k$ -certified (or has a  $k$ -certificate) if for every path  $P = u \rightarrow v$  in  $G[V \setminus \{s\}]$  we have

$$d(v) \leq (1 + \xi)^k \cdot (d(u) + w(P)).$$

As there exist no paths from  $s$  in  $G[V \setminus \{s\}]$ ,  $s$  is trivially  $k$ -certified for any  $k$ . We now state some properties vertex certificates’ behavior after running Propagate.

Observation 4.4. Let  $V_{\text{input}} \subseteq V$  be an arbitrary subset. Suppose  $V_{\text{touched}} := \text{Propagate}(V_{\text{input}})$  is run. Then, for every  $u \in V \setminus V_{\text{touched}}$ , if  $u$  was  $k$ -certified before, then it remains  $k$ -certified.

---

<sup>7</sup>This follows by the inequality  $(1 + x/(2k))^k \leq e^{x/2} \leq 1 + x$  that holds for all  $x \in (0, 1)$ .



Proof. **TOPROVE 2** □

Lemma 4.5. Suppose ?? is satisfied. Let  $U \subseteq V$  be some set of  $k$ -certified vertices. After running  $V_{\text{touched}} := \text{Propagate}(V \setminus U)$ , the vertices  $V_{\text{touched}}$  are  $(k + 1)$ -certified.

Proof. **TOPROVE 3** □

The pursued data structure additionally tracks information about vertex certificates. Since that a vertex  $v$  is  $k$ -certified implies it is  $(k + 1)$ -certified as well, for each vertex  $v \in V$  we will only care about the smallest value of  $k$  that we can deduce. Initially (before updates come), each vertex is 0-certified since all edges are relaxed. Note that due to an insertion of an edge from  $s$  itself alone, no vertex stops being  $k$ -certified as ?? is concerned about paths in  $G[V \setminus \{s\}]$ . Moreover, if an insertion of an edge is handled as in ??, then by ??, a vertex  $u \in V$  can only stop being  $k$ -certified when  $u \in V_{\text{touched}}$  after running Propagate. We will show that ?? is useful for controlling this effect and keeping vertex certificates small.

For each  $v \in V$ , the data structure additionally maintains an integer  $\text{rank}(v) \geq 0$ , such that  $v$  is  $\text{rank}(v)$ -certified. Initially,  $\text{rank}(v) = 0$  for all vertices  $v \in V$ . Let  $C_k \subseteq V$  denote the current set of vertices  $v$  with  $\text{rank}(v) = k$  and let  $r$  denote the current maximum rank. The data structure explicitly maintains these sets alongside the information about their total degree  $\deg(C_k) = \sum_{v \in C_k} \deg(v)$ . This additional bookkeeping is easy to implement subject to vertices' ranks changes and will be mostly neglected in the following description and analysis.

As discussed previously, running Propagate in line ?? of ?? invalidates the (at most)  $r$ -certificates of vertices from  $V_{\text{touched}}$ . However, by ??, for each  $v \in V_{\text{touched}}$  we can fix this by assigning  $\text{rank}(v) := r + 1$  which increases the maximum rank  $r$  by one. Doing this alone would cause  $r$  to grow linearly with the number of Propagate invocations made during the algorithm and consequently make the distance estimates very inaccurate. In order to negate this effect, we will use the Synchronize procedure (called in line ?? in ??) shown in ??.

---

Algorithm 3 Synchronize()

---

- 1: while there exists  $k \in \{0, \dots, r\}$  such that  $\deg(C_k) \leq \deg(C_{k+1}) + \dots + \deg(C_r)$  do
  - 2:      $V_{\text{touched}} := \text{Propagate}(C_k \cup C_{k+1} \cup \dots \cup C_r)$
  - 3:     Set the ranks of vertices  $V_{\text{touched}}$  to  $k$
- 

Observation 4.6. Synchronize correctly reassigns ranks.

Proof. **TOPROVE 4** □

Lemma 4.7. When Synchronize completes, the maximum rank  $r$  satisfies  $r \leq \log_2 m$ .

Proof. **TOPROVE 5** □

?? and ?? show that by using synchronization, the ranks of the vertices will be correct and will be kept bounded by  $\log_2 m$ .

Lemma 4.8. For any  $t \in V$ , we have  $d(t) \leq (1 + \xi)^{r+1} \cdot \text{dist}(s, t) = (1 + \xi)^{1+\log_2 m} \cdot \text{dist}(s, t)$ .

Proof. **TOPROVE 6** □



Running time analysis. First of all, observe that all edge updates that do not immediately decrease some estimate by a factor of at least  $1+\xi$  are processed in constant time. The total cost of processing such updates is hence  $O(\Delta)$ .

It is easy to see that, apart from the above, the running time is dominated by the total time of the Propagate calls. In fact, the total update time can be (coarsely) bounded by summing the degrees of the vertices that are pushed to the queue  $Q$  in Propagate. Similarly as in ??, we can bound the total cost incurred by pushing a vertex after its estimate drops by

$$O(m\ell + n\ell \log n) = O((m + n \log n) \log(nW)/\xi).$$

Note that the only other way a vertex  $v$  can be pushed to the queue in  $Q$  (without a drop in the estimate  $d(v)$ ) is if  $v$  is in the input set of the Propagate call inside Synchronize. Thus, consider some call  $\text{Propagate}(C_k \cup C_{k+1} \cdots \cup C_r)$  inside Synchronize for  $k \in \{0, \dots, r\}$  such that  $\deg(C_k) \leq \deg(C_{k+1}) + \cdots + \deg(C_r)$ . Observe that in such a case we can bound the cost of processing the vertices initially pushed to the queue  $Q$  by  $O(\log n)$  times

$$\deg(C_k) + \deg(C_{k+1}) + \cdots + \deg(C_r) \leq 2(\deg(C_{k+1}) + \cdots + \deg(C_r)).$$

Recall that after such a Propagate call completes, the ranks of all vertices  $C_{k+1} \cup \dots \cup C_r$  drop by at least 1. We can thus bound the total sum of expressions  $\deg(C_{k+1}) + \dots + \deg(C_r)$  throughout by  $m$  times the maximum number of times some vertex  $v$  may have its rank increased. However, note that the rank of a vertex  $v$  can only increase if  $v \in V_{\text{touched}}$  in line ?? of ?. But the presence of  $v$  in  $V_{\text{touched}}$  in that case is caused by  $d(v)$  dropping by a factor of at least  $1+\xi$ . We conclude that  $\text{rank}(v)$  may only increase  $O(\ell)$  times. As a result, the total cost Propagate calls in Synchronize can also be bounded by  $O(m\ell \log n) = O(m \log(nW) \log(n)/\xi)$ .

The above analysis combined with ?? yields ??.

### 4.3 Handling general edge insertions in a special case

Recall that the data structure of ?? does not allow edge insertions except for edges that originate in the source. In our later developments (??), we need to insert batches  $F$  of arbitrary edges to the data structure from time to time. One way to handle this would be to reinitialize the data structure for the graph  $G + F$  after each such insertions batch. This would be too costly to yield non-trivial applications though.

We nevertheless prove that if an additional assumption holds that the maintained estimates for  $G$  are reasonably accurate for  $G + F$  before applying the insertion, then inserting  $F$  can be handled without compromising the running time albeit at the cost of introducing additional error.

**Lemma 4.9.** The data structure of ?? can be extended to also support inserting batches  $F \subseteq V \times V$  of arbitrary weighted edges to  $G$  if provided with an integer exponent  $\alpha \geq 0$  such that the currently stored estimates satisfy  $d(v) \leq (1 + \xi)^\alpha \text{dist}_{G+F}(s, v)$  for all  $v \in V$ .

Then, at all times the maintained estimates satisfy:

$$d(v) \leq (1 + \xi)^{1+\rho+\log_2 m} \text{dist}_G(s, v) \text{ for all } v \in V,$$

where  $\rho$ , called the rank offset, equals the exponent  $\alpha$  of the most recently performed batch-insertion. The total update time of the data structure remains  $O(m \log(n) \log(nW)/\xi + \Delta)$ .

One can view the basic data structure of ?? (supporting only source insertions) as keeping the rank offset  $\rho$  defined as above always zero.

To prove ??, we start with the following lemma.

Lemma 4.10. Let  $F \subseteq V \times V$  be some set of weighted edges. Suppose for some integer  $\alpha \geq 0$ , the maintained estimates  $d : V \rightarrow \{0\} \cup [1, nW]$  for  $G$  additionally satisfy

$$d(v) \leq (1 + \xi)^\alpha \text{dist}_{G+F}(s, v) \quad (1)$$

for all  $v \in V$ . Then, after inserting the edges  $F$  one by one according to ??, ?? is satisfied and every vertex is  $\alpha$ -certified.

Proof. **TOPROVE 7** □

Applying ?? with  $\alpha \geq 1$  cannot guarantee that ?? holds after inserting a batch  $F$ , as some of the vertices might have had their rank smaller than  $\alpha$  before. As a result, their ranks might be incorrect now. To deal with this, we slightly relax the invariant posed on the maintained ranks. Specifically, we augment the data structure of ?? to also maintain an integral rank offset  $\rho \geq 0$  such that every vertex  $v \in V$  is  $(\rho + \text{rank}(v))$ -certified. One can view the basic data structure of ?? as keeping the rank offset defined like this always zero. This way, whenever ?? is applied given some exponent  $\alpha$ , we can simply reset  $\rho := \alpha$ . It is easy to verify that both ?? and ?? remain valid after this change. However, the bound in ?? gets weakened to

$$d(t) \leq (1 + \xi)^{\rho+r+1} \cdot \text{dist}(s, t) \leq (1 + \xi)^{\rho+\log_2 m+1} \cdot \text{dist}(s, t). \quad (2)$$

The running time analysis of the data structure of ?? goes through unaltered. Therefore, using ??, we obtain ??.

#### 4.4 Resetting the data structure

Using ?? repeatedly may lead to a significant growth of the rank offset which makes the vertex certificates larger and thus the maintained estimates less and less accurate. Later on, in our incremental APSP application (??), this will indeed prove to be a problem and we will need to keep the rank offset bounded. In this section we discuss some ways to achieve that.

A deterministic recompute-from-scratch reset. Recall from ?? that running  $\text{Propagate}(V)$  makes all vertices 0-certified. As a result, by running  $\text{Propagate}(V)$  and subsequently setting all vertex ranks and the rank offset  $\rho$  to 0, in  $O(m + n \log n)$  time we can get rid of the pathwise error of the maintained estimates  $d(\cdot)$ .

Note that in the running time analysis so far, the work performed inside  $\text{Synchronize}$  was fully charged to the unit rank increases that made the ranks of the  $\text{Propagate}$ 's input vertices positive. Consequently, as the discussed reset procedure zeroes out ranks, invoking it any number of times interleaved with the other discussed operations does not increase the total update time bound of the other operations of the data structure.

Randomized reset. Reducing the vertex certificates all the way down to 0 might be unnecessary; keeping the certificates bounded might be just enough. In ??, we discuss the randomized approach of [?] used to achieve that (adapted to our needs). Formally, we show the following:

Theorem 4.11. Let  $\lambda \in [1, \ell]$  be an integer. Then, in  $O(m \log^2(nW) \log^2(n) / (\lambda^3 \xi^2))$  additional time one can adjust the estimates so that every vertex is  $\lambda$ -certified with high probability.

## 5 Incremental all-pairs shortest paths for adaptive adversaries

In this section, we describe our incremental APSP data structure. We prove:

Theorem ?? . Let  $G = (V, E)$  be a digraph with edge weights in  $\{0\} \cup [1, W]$  and let  $\epsilon \in (0, 1)$ . There exist data structures explicitly maintaining distance estimates  $d : V \times V \rightarrow \mathbb{R}_{\geq 0}$  such that

$$\text{dist}_G(u, v) \leq d(u, v) \leq (1 + \epsilon) \cdot \text{dist}_G(u, v)$$

for all  $u, v \in V$ , subject to edge insertions or weight decreases issued to  $G$ :

- A deterministic one with  $O(m^{3/2}n^{3/4}\log^2(nW)\log(n)/\epsilon^{3/2} + \Delta)$  total update time.
- A Monte Carlo randomized one that produces correct answers against an adaptive adversary (whp.) with  $O(m^{4/3}n^{5/6}\log^3(nW)\log(n)/\epsilon^{7/3} + \Delta)$  total update time.

Here,  $m$  denotes the final number of edges in  $G$  and  $\Delta$  the total number of updates issued.

Let  $b \in [1, m]$  be an integer parameter to be set later. The data structure operates in phases of  $b$  edge updates. While a phase proceeds, denote by  $E_{\text{cur}}$  the edges whose insertions or weight decreases have been issued in the current phase. Let  $V_{\text{cur}}$  be the set of endpoints of  $E_{\text{cur}}$ , so that  $|V_{\text{cur}}| \leq 2b$ . Moreover, let  $G_{\text{beg}}$  denote the graph  $G$  from the beginning of the phase, that is, with all insertions from the previous phases applied. Note that  $G = G_{\text{beg}} + E_{\text{cur}}$  at all times.

Let  $p \in \mathbb{Z}_+$  be a reset parameter and  $\xi \in (0, 1)$  be an accuracy parameter, both also to be set later. Roughly speaking, the accuracy of the data structure will deteriorate slightly after each phase, and will be restored every  $p$  phases. The accuracy parameter will be shared by all the data structure's components and will be adjusted based on  $\epsilon$  and other parameters at the very end.

We will now describe the data structure's components and the interplay between them. For convenience, we will use the following notation: if  $\mathcal{C}$  is a component maintaining some vertex estimates, we will write  $\mathcal{C}(v)$  to denote the estimate that  $\mathcal{C}$  maintains for the vertex  $v$ .

Before we continue, let us refer to a standard way of reducing to the case when there is only at most  $O(m \log(W)/\epsilon)$  weight decreases in total (see, e.g., [?]). Indeed, we may only record a weight decrease for some edge  $e \in E$  if the new weight of  $e$  is smaller than the previously recorded weight of  $e$  by a factor of at least  $1 + \epsilon$ , and ignore it otherwise. Having done that, by proceeding normally afterwards, the returned distance estimates might be distorted by an extra  $1 + \epsilon$  factor (in addition to the  $1 + \epsilon$  factor we aim for). But this can be easily dealt with at no asymptotic cost by decreasing  $\epsilon$  by a constant factor (say 4) in the very beginning. Filtering weight decreases like that clearly costs only  $O(\Delta)$  additional time. Note that with this updates filtering scheme applied, we can bound the number of phases by  $O(m \log(W)/(b\epsilon))$ .

SSSP data structures with shortcuts. For each  $s \in V$ , we maintain a data structure  $\mathcal{D}_s$  of ?? extended to support arbitrary batch insertions as described in ??. The data structure  $\mathcal{D}_s$  maintains a graph  $G_s$  obtained from  $G_{\text{beg}}$  by extending it with  $n$  shortcut edges  $e_{s,u} = su$ . To the best of our knowledge, the idea of using shortcut edges for partially dynamic APSP is due to [?] and has been applied in the incremental data structure of [?] as well.

The weight of a shortcut edge  $e_{s,u}$  always corresponds to the length of some  $s \rightarrow u$  walk in the current graph  $G$ . Hence,  $w_{G_s}(e_{s,u}) \geq \text{dist}_G(s, u)$ . As a result, for each  $v \in V$ , we have

$$\text{dist}_G(s, v) \leq \text{dist}_{G_s}(s, v) \leq \text{dist}_{G_{\text{beg}}}(s, v).$$

Our goal will be to maintain the shortcuts so that after each update, the following holds:

$$\text{dist}_{G_s}(s, v) \leq (1 + \xi)^{O(p \log n)} \cdot \text{dist}_G(s, v).$$

For an appropriate  $\xi$ , we will set  $d(u, v) := \mathcal{D}_u(v)$  to be the final outputs of our data structure.

Recall that using the extension (??) requires taking into account the rank offset  $\rho$  defined therein. All our single-source data structures will use the same rank offset  $\rho$  which will grow slightly after every phase, but at the same time it will be kept under control using resets.

We also maintain a symmetric collection of data structures of ?? on the reverse graph  $G^R$ , i.e.,  $G$  with edge directions reversed. That is, a data structure  $\mathcal{D}_t^R$  maintains  $G_{\text{beg}}^R$  with some shortcuts  $e_{t,u}^R$  corresponding to paths from  $t$  in  $G^R$  added. The actual purpose of  $\mathcal{D}_t^R$  is to maintain approximate shortest paths to the vertex  $t$  in  $G$  from all  $s \in V$ ; running a data structure on the reverse graph turns such a single-sink problem into a single-source problem.

The single-source data structures  $\mathcal{D}_s$  and  $\mathcal{D}_t^R$  are shared by all the phases.

APSP between insertion endpoints. Fix some phase. Let  $E_{\text{cur}}(u, v)$  denote the smallest weight of an edge  $uv \in E_{\text{cur}}$ , or  $\infty$  if no such edge exists in  $E_{\text{cur}}$ . Let  $H$  be a complete digraph on  $V_{\text{cur}}$  such that for all  $u, v \in V_{\text{cur}}$ , we have  $w_H(uv) := \min(\mathcal{D}_u(v), E_{\text{cur}}(u, v))$ .

Note that while the phase proceeds,  $V_{\text{cur}}$  grows and thus the graph  $H$  also grows. Moreover, the weights of  $H$  undergo weight decreases while the estimates in the data structures  $\mathcal{D}_u$ , where  $u \in V_{\text{cur}}$ , change. We store the graph  $H$  using the near-optimal incremental APSP data structure  $\mathcal{A}$  for dense graphs, formally characterized below.

Theorem 5.1. [?] Let  $\xi \in (0, 1)$ . Let  $G = (V, E)$  be a directed graph with edge weights in  $\{0\} \cup [1, W]$ . There exists a deterministic data structure maintaining  $G$  subject to edge insertions and weight decreases explicitly maintaining for all pairs  $u, v \in V$  an estimate  $d(u, v)$  such that

- $d(u, v)$  is the weight of some  $u \rightarrow v$  path in  $G$ , ie.,  $\text{dist}_G(u, v) \leq d(u, v)$ ,
- $d(u, v) \leq (1 + \xi)\text{dist}_G(u, v)$ .

The total update time is  $O(n^3 \log(n) \log(nW)/\xi + \Delta)$ , where  $\Delta$  is the total number of updates issued.

Remark 5.2. The above data structure [?, Lemma 5.4], stated originally, does not enforce the returned estimates to be lengths of actual paths in  $G$ . However, it can be easily modified to maintain “witnesses”, i.e., weights of paths  $u \rightarrow v$  of weight smaller than the corresponding estimate  $d(u, v)$ .

Remark 5.3. The data structure in ?? assumes a fixed vertex set  $V$ , whereas in our use case, the vertex set grows. However, we know a bound  $2b$  on the final size of  $V_{\text{cur}}$  so we can start the data structure of ?? with a vertex set of size  $2b$  and map the remaining placeholder vertices to new vertices entering  $V_{\text{cur}}$  while the phase proceeds.

The data structure  $\mathcal{A}$  is reinitialized from scratch each time a new phase starts. In the following, we will write  $\mathcal{A}(u, v)$  to denote the estimate  $d(u, v)$  maintained by the data structure  $\mathcal{A}$ .

Setting the shortcuts’ weights. We are now ready to describe how the weights of the shortcuts in the data structures  $\mathcal{D}_s$  and  $\mathcal{D}_s^R$  are defined and updated.

We maintain the following invariants:

- (1)  $w(e_{s,t}) \leq \mathcal{A}(s, t)$  and  $w(e_{t,s}^R) \leq \mathcal{A}(s, t)$  for every  $s, t \in V_{\text{cur}}$ ,
- (2)  $w(e_{s,t}) \leq \mathcal{D}_t^R(s)$  for every  $s \in V$  and  $t \in V_{\text{cur}}$ .
- (3)  $w(e_{t,s}^R) \leq \mathcal{D}_s(t)$  for every  $s \in V_{\text{cur}}$  and  $t \in V$ .

Let us explain how invariant (1) above is maintained; the other are maintained analogously. Whenever the data structure  $\mathcal{A}$  changes some of its maintained estimates  $\mathcal{A}(u, v)$ , the change is passed as a source-weight-decrease to the data structures  $\mathcal{D}_u$  and  $\mathcal{D}_v^R$ . The existing weight of the relevant shortcut therein might be already smaller; in such a case the weight decrease is simply ignored.

**Closing a phase.** When a phase is completed, the edges  $E_{\text{cur}}$  are batch-inserted to  $\mathcal{D}_s$  and  $\mathcal{D}_s^R$  for all  $s \in V$  using Lemma ?? . For this to be allowed, estimates in these data structures (storing the graphs  $G_{\text{beg}}$  augmented with some shortcuts) need to approximate the distances in  $G$  well. We will later prove (??) that this is the case indeed. Moreover, we will show that the rank offset of the single-source data structures will only grow by  $O(\log m)$  as a result of these batch-insertions.

**Resets.** Every  $p$  phases, we perform a reset in all the maintained single-source data structures, as described in Section ?? . In the deterministic variant of our data structure, we use the deterministic reset, which reduces the parameter  $\rho$  in each of them all the way to 0. In the randomized variant, we use randomized resetting (??) to reduce the rank offset  $\rho$  to  $\rho^* := \lceil p \log_2 m \rceil$ .

### 5.1 Estimate error analysis

Let  $k < p$  denote the number of phases completed since the last reset. In this section, we bound the errors of the individual components of the data structure as a function of  $k$ .

Let  $y := \log_2 m + 1$ . The single-source data structures  $\mathcal{D}_s$  and  $\mathcal{D}_s^R$  (for  $s \in V$ ) will all use the rank offset  $\rho$  no more than  $\rho^* + k \cdot 4y$ . Note that for  $k = 0$ , that is, immediately after a reset, this is indeed achieved by construction. The below lemma bounding the distortions of the estimates maintained in individual components is the key to the correctness.

**Lemma 5.4.** Let  $k < p$  denote the number of phases completed since the last reset. Then:

- (i) for all  $s, t \in V$ ,  $\mathcal{D}_s(t) \leq (1 + \xi)^{\rho^* + k \cdot 4y} \cdot \text{dist}_{G_{\text{beg}}}(s, t)$ ,
- (ii) for all  $s, t \in V$ ,  $\mathcal{D}_t^R(s) \leq (1 + \xi)^{\rho^* + k \cdot 4y} \cdot \text{dist}_{G_{\text{beg}}}(s, t)$ ,
- (iii) for all  $s, t \in V_{\text{cur}}$ ,  $\mathcal{A}(s, t) \leq (1 + \xi)^{\rho^* + k \cdot 4y + y} \cdot \text{dist}_G(s, t)$ ,
- (iv) for all  $s \in V_{\text{cur}}$  and  $t \in V$ ,  $\mathcal{D}_s(t) \leq (1 + \xi)^{\rho^* + k \cdot 4y + 2y} \cdot \text{dist}_G(s, t)$ ,
- (v) for all  $s \in V$  and  $t \in V_{\text{cur}}$ ,  $\mathcal{D}_t^R(s) \leq (1 + \xi)^{\rho^* + k \cdot 4y + 2y} \cdot \text{dist}_G(s, t)$ ,
- (vi) for all  $s, t \in V$ ,  $\mathcal{D}_s(t) \leq (1 + \xi)^{\rho^* + k \cdot 4y + 3y} \cdot \text{dist}_G(s, t)$ ,
- (vii) for all  $s, t \in V$ ,  $\mathcal{D}_t^R(s) \leq (1 + \xi)^{\rho^* + k \cdot 4y + 3y} \cdot \text{dist}_G(s, t)$ .

**Proof.** TOPROVE 8 □

By items (vi) and (vii) of ?? and since  $\rho^* + k \cdot 4y + 4y = \rho^* + (k + 1) \cdot 4y$ , at the end of the phase one can indeed insert the edges  $E_{\text{cur}}$  to all the data structures  $(\mathcal{D}_s)_{s \in V}$  and  $(\mathcal{D}_t^R)_{t \in V}$  using ??, set  $\rho := \rho^* + (k + 1) \cdot 4y$ , and initialize the next phase.

By ??, we conclude that that the estimates stored in data structures  $(\mathcal{D}_s)_{s \in V}$  satisfy:

$$\mathcal{D}_s(t) \leq (1 + \xi)^{\rho^* + 4p \cdot y} \cdot \text{dist}_G(s, t). \leq (1 + \xi)^{5p \cdot y} \cdot \text{dist}_G(s, t).$$

Thus, as long as  $\xi \leq \frac{\epsilon}{10py}$ , they indeed constitute  $(1 + \epsilon)$ -approximate distance estimates in  $G$ .

## 5.2 Running time analysis

First of all, by ?? and ??, the total cost of setting up and maintaining the single-source data structures  $(\mathcal{D}_s)_{s \in V}$  and  $(\mathcal{D}_t^R)_{t \in V}$  through all updates, excluding the resets and the  $\Delta$  terms denoting the numbers of updates issued to these data structures is  $O(n \cdot m \log(nW) \log(n)/\xi)$ .

For each of the  $O(m \log(W)/(\epsilon b))$  phases, we reinitialize and run a fresh data structure of ?? on a graph with  $O(b)$  vertices and  $O(b^2)$  edges. The total update time of that data structure, excluding the  $\Delta$  term counting the issued updates, is  $O(b^3 \log(nW) \log(n)/\xi)$  per phase. Hence, the total cost incurred through all phases is  $O(mb^2 \log^2(nW) \log(n)/(\epsilon \xi))$ .

Now, let us consider the number of updates issued to the maintained components. Recall our assumption that the total number of updates issued to  $G$  is  $O(m \log(nW)/\epsilon)$ . Consequently, the total number of “non-shortcut” updates issued to the single-source data structures is  $\tilde{O}(nm \log(nW)/\epsilon)$ . Similarly, the number of updates issued to the data structures of ?? that are not caused by some estimate change in the single-source data structures is  $O(m \log(nW)/\epsilon)$ .

Every update of a “shortcut” edge in the maintained data structures is caused by an estimate change in another component. Hence, the total number of shortcut edge updates can be bounded by the total number of estimate changes in all the maintained components. That quantity can be bounded in turn by the total update time of all the maintained components ignoring the  $\Delta$  terms denoting the numbers of updates issued. We conclude that this additional cost is asymptotically no more than what we have taken into account so far.

Recall that in the deterministic variant, a reset costs  $O(mn \log n)$  time. In the randomized variant, by ?? applied with  $\lambda = \rho^*$ , a reset costs  $O\left(\frac{nm \log^2(nW)}{p^3 \xi^2 \log n}\right)$  time. Recall that the number of resets is  $O(m \log(W)/(\epsilon bp))$ , so the total cost of all resets is  $O(m^2 n \log(W) \log(n)/(\epsilon bp))$  in the deterministic variant and  $O\left(\frac{m^2 n \log^3(nW)}{\epsilon b p^4 \xi^2 \log n}\right)$  in the randomized variant. By using  $\xi := \epsilon/10p(\log m + 1)$ , the final running time in the deterministic case becomes:

$$O\left(nm \log(nW)/\epsilon + nmp \log(nW) \log^2(n)/\epsilon + mb^2 p \log^2(nW) \log^2(n)/\epsilon^2 + \frac{m^2 n \log(n) \log(W)}{\epsilon \cdot b \cdot p}\right).$$

By setting  $b = \sqrt{n}$  and  $p = \frac{m^{1/2} \epsilon^{1/2}}{n^{1/4} \log n}$ , the running time becomes  $O(m^{3/2} n^{3/4} \log^2(nW) \log(n)/\epsilon^{3/2})$ .

In the randomized case, the last term is replaced with  $m^2 n \log^3(nW) \log n/(bp^2 \epsilon^3)$ . By setting  $b = \sqrt{n}$  and  $p = \frac{m^{1/3}}{n^{1/6} \epsilon^{1/3}}$ , the running time becomes  $O(m^{4/3} n^{5/6} \log^3(nW) \log(n)/\epsilon^{7/3})$ .

## 6 Offline incremental SSSP in near-linear time

In this section we describe a simple near-optimal offline incremental  $(1 + \epsilon)$ -approximate SSSP data structure and prove the following:

**Theorem ??.** Let  $\epsilon \in (0, 1)$ . Let  $G = (V, E)$  be a digraph with edge weights in  $\{0\} \cup [1, W]$  and a source  $s \in V$ . Suppose we are given a sequence of  $\Delta$  incremental updates to  $G$  (edge insertions or weight decreases). Let  $G^0, \dots, G^\Delta$  denote the subsequent versions of the graph  $G$ .

Then in  $O(m \log(n) \log(nW) \log^2(\Delta)/\epsilon + \Delta)$  deterministic time one can compute a data structure supporting queries  $(v, j)$  about a  $(1 + \epsilon)$ -approximate estimate of  $\text{dist}_{G^j}(s, v)$ . The query time is  $O(\log(\log(nW) \log(\Delta)/\epsilon))$ . For example, if  $\epsilon, W, \Delta \in \text{poly}(n)$ , the query time is  $O(\log \log n)$ .

**Main idea.** We first give a quick overview of the approach. We construct a collection  $\mathcal{D}$  of distance estimates found for different vertices and different versions of the graph. Assuming that for some

---

**Algorithm 4** Search( $\alpha, \beta$ )

---

Let  $\mathcal{D}$  be a global set that stores some estimates of  $\text{dist}_{G^j}(s, u)$  for different  $u$  and  $j$ .

Let  $d_{\leq \alpha}(u)$  denote the smallest estimate of  $\text{dist}_{G^j}(s, u)$  stored in  $\mathcal{D}$  for any  $j \leq \alpha$ .

Let  $d_{\geq \beta}(u)$  denote the largest estimate of  $\text{dist}_{G^j}(s, u)$  stored in  $\mathcal{D}$  for any  $j \geq \beta$ .

- 1: If  $[\alpha, \beta]$  is empty, end the procedure
  - 2:  $X := \{u \in V : d_{\leq \alpha}(u) > (1 + \xi)d_{\geq \beta}(u)\}$
  - 3:  $\gamma := \lfloor (\alpha + \beta) / 2 \rfloor$
  - 4: Construct a graph  $H$  equal to  $G^\gamma[X]$  with an additional source vertex  $f$
  - 5: for  $v \in X$  do
  - 6:      $d_{\text{init}}(v) := \min \{d_{\leq \alpha}(u) + w(e) : uv = e \in E(G^\gamma), u \notin X\}$
  - 7:     In  $H$  connect  $f$  to  $v$  with an edge of weight  $d_{\text{init}}(v)$
  - 8:  $d_\gamma := \text{Dijkstra}(H, f)$   $\triangleright d_\gamma(v) = \text{dist}_H(f, v)$  for all  $v$
  - 9: for  $v \in X$  do
  - 10:     Store  $d_\gamma(v)$  in  $\mathcal{D}$  as an approximation of  $\text{dist}_{G^\gamma}(s, v)$
  - 11: Search( $\alpha, \gamma - 1$ )
  - 12: Search( $\gamma + 1, \beta$ )
- 

query  $(u, j)$  we store an estimate of  $\text{dist}_{G^j}(s, u)$  in  $\mathcal{D}$ , we will be able to answer that query directly. If the answer cannot be retrieved from  $\mathcal{D}$  directly, we look for the smallest estimate of  $\text{dist}_{G^i}(s, u)$  stored for some  $i < j$  that we can find in  $\mathcal{D}$  and return that value as the answer instead. If we ensure that for each such  $j$  we can also find in  $\mathcal{D}$  an estimate of  $\text{dist}_{G^k}(s, u)$ , where  $k > j$  that is “close enough” to the reported estimate of  $\text{dist}_{G^j}(s, u)$ , we can use the two “surrounding” estimates as lower and upper bounds for the distance sought in the query.

We build the collection  $\mathcal{D}$  outlined above as follows. We first find exact distances in the initial and final versions  $G^0$  and  $G^\Delta$  and store them in  $\mathcal{D}$ . We then use a divide-and-conquer approach. We use a recursive Search subroutine (??), which given the current set  $\mathcal{D}$  and a range  $[\alpha, \beta]$ , determines the set of vertices  $u$  such that can estimate  $\text{dist}_{G^j}(u)$  for all  $j \in [\alpha, \beta]$  using the estimates already stored, and searches for additional estimates for the remaining vertices.

**Setup.** Similarly as with the previous data structures, let us assume for simplicity that  $G$  initially contains  $n$  edges  $su$  of weight  $nW$ , so that  $\text{dist}(s, u) \leq nW$  for all vertices  $u$  throughout all updates. Let  $\xi$  be an accuracy parameter, set to  $\epsilon / (2 \log_2(\Delta) + 2)$ . We will write  $d_{\leq \alpha}(u)$  to denote the smallest estimate of  $\text{dist}_{G^j}(s, u)$  stored in  $\mathcal{D}$  for some  $j \leq \alpha$  and similarly write  $d_{\geq \beta}(u)$  to denote the largest estimate of  $\text{dist}_{G^j}(s, u)$  stored in  $\mathcal{D}$  for some  $j \geq \beta$ . When using this notation we consider all the estimates that are currently stored in  $\mathcal{D}$ ; what “currently” means depends on the context.

## 6.1 The algorithm

We start by storing in  $\mathcal{D}$  the exact value of  $\text{dist}_{G^0}(s, u)$  and  $\text{dist}_{G^\Delta}(s, u)$  for each vertex  $u$ . We find them by running Dijkstra’s algorithm on  $G^0$  and  $G^\Delta$ . We then call the procedure Search( $0, \Delta$ ), which accesses  $\mathcal{D}$  and stores some additional distance estimates of  $\text{dist}_{G^j}(s, u)$  for different  $j$  and  $u$  (see ??). After that, the construction of  $\mathcal{D}$  is complete and we can process the queries.

For each query concerning  $\text{dist}_{G^j}(s, u)$  we return the value  $d_{\leq j}(u)$ , which corresponds to the smallest estimate of  $\text{dist}_{G^i}(s, u)$  for some  $i \leq j$  stored in  $\mathcal{D}$ .



## 6.2 Correctness analysis

In this section we analyze the error buildup of the estimates stored in  $\mathcal{D}$  by different Search calls, depending on their depth in the recursion tree, where we assume that the depth of the Search(0,  $\Delta$ ) call that we invoke directly is 0. We then bound the error of the reported answers.

Lemma 6.1. Consider a call Search( $\alpha, \beta$ ) at depth  $h$ . Then for every  $t \in X$ , we have

$$\text{dist}_{G^\gamma}(s, t) \leq d_\gamma(t) \leq (1 + \xi)^{h+1} \cdot \text{dist}_{G^\gamma}(s, t).$$

Proof. **TOPROVE 9** □

Now observe that the depth of every (not immediately ending) recursive call is at most  $\log_2(\Delta)$  since the interval  $[\alpha, \beta]$  is halved in children calls. By putting  $\xi := \epsilon/(2\log_2(\Delta) + 2)$ , the approximation factor for every estimate stored in  $\mathcal{D}$  is  $1 + \epsilon$  by reasoning similar to ??.

Corollary 6.2. For any query  $(u, j)$ ,  $d_{\leq j}(u)$  is a  $(1 + \epsilon)$ -approximate estimate of  $\text{dist}_{G^j}(s, u)$ .

Proof. **TOPROVE 10** □

## 6.3 Running time analysis

We first describe some lower-level details of the implementation. Starting off, the set  $\mathcal{D}$  can be implemented as a collection of balanced binary search trees. For each vertex  $u$ , we have a separate balanced BST that stores the set of approximations of  $\text{dist}_{G^j}(s, u)$  for different timestamps  $j$ , sorted in order of increasing  $j$ . We can thus perform insertions and queries for the largest/smallest approximation stored for a given range of timestamps in time logarithmic to the number of stored elements. Later on, we prove that the size of each stored BST is  $O(\log(nW) \log^2(\Delta)/\epsilon)$ . Therefore, all high-level operations involving  $\mathcal{D}$  take  $O(\log(\log(nW) \log(\Delta)/\epsilon))$  time.

Secondly, observe that the bottleneck of the cost of a call Search (??) is running Dijkstra's algorithm in line ??. First, we need to construct the set  $X$  efficiently in line ??. It is too costly to do that naively by iterating through all vertices. Thankfully, observe that any vertex  $v$  may belong to the set  $X$  only if it belonged to the set  $X$  of the parent procedure call, if such a call exists. Therefore, we could use a slightly modified algorithm, where we pass the set  $X$  to the children calls as  $X_0$ , so that when constructing respective sets  $X$  in children calls, we only iterate through  $X_0$ . Then, the total cost of computing all the sets  $X$  can be easily seen to be proportional to their total size times the cost of checking the condition  $d_{\leq \alpha}(u) > (1 + \xi)d_{\geq \beta}(u)$ .

The key claim that we prove to argue about the efficiency of the proposed preprocessing is that each vertex may belong to the set  $X$  of only  $O(\log(nW) \log(\Delta)/\xi)$  recursive calls of Search.

Fix some vertex  $u \in V$ . If  $u$  belongs to the set  $X$  inside some Search call, we say that the call is costly for  $u$ . Our strategy is to bound the number of costly calls at any fixed depth of the recursion. Formally, we say that the two recursive Search calls are the 1-descendants of the call that invoked them. Inductively, for any integer  $h > 1$ , the  $h$ -descendants of some Search call are the  $(h - 1)$ -descendants of its 1-descendants. For convenience, we also say that any call is its own 0-descendant. We can now finally state our main lemma in this section.

Lemma 6.3. For any  $h \geq 0$  and any call Search( $\alpha, \beta$ ) that was invoked at some point of the algorithm, the number of its costly  $h$ -descendants is at most

- 1°) 0 if  $d_{\leq \alpha}(u) = 0$ ,
- 2°)  $1 + \log_{1+\xi}(d_{\leq \alpha}(u))$  if  $d_{\leq \alpha}(u) > 0$  and  $d_{\geq \beta}(u) = 0$ ,

3°)  $\log_{1+\xi} \left( \frac{d_{\leq \alpha}(u)}{d_{\geq \beta}(u)} \right)$  if  $d_{\leq \alpha}(u) > 0$  and  $d_{\geq \beta}(u) > 0$ .

Proof. **TOPROVE 11** □

Applying the above theorem to the call  $\text{Search}(0, \Delta)$ , we can bound the total number of costly calls for a given vertex  $u$  by  $(1 + \log_{1+\xi}(nW)) \cdot \log_2 \Delta$ , which after substituting  $\xi = \Theta(\epsilon/\log \Delta)$  is  $O(\log(nW) \log^2(\Delta)/\epsilon)$ . The total cost of the Dijkstra invocations (line ?? of ??), combined with the total number of Search calls, is  $O(\Delta \log(\Delta) + m \log(n) \log(nW) \log^2(\Delta)/\epsilon)$ .

Recall from ?? that for large values of  $\Delta = \Omega(m \log(W)/\epsilon)$ , we can reduce our problem to the case where we have at most  $O(m \log(W)/\epsilon)$  weight decreases in total by filtering the updates in  $O(\Delta)$  time at the start. The time complexity thus becomes  $O(m \log(n) \log(nW) \log^2(\Delta)/\epsilon + \Delta)$ .

## 7 Randomized reset

This section is devoted to proving ?. Below we state the procedure Randomized-Reset that achieves the goal if called with parameter  $\lambda$ . After the procedure completes, we simply reset the rank offset  $\rho$  to  $\lambda$ . Provided that the procedure makes every vertex  $\lambda$ -certified, any particular vertex  $v$  is clearly  $(\rho + \text{rank}(v))$ -certified afterwards.

---

Algorithm 5 Randomized-Reset( $\lambda$ )

---

```

1:  $Z := \emptyset$ 
2: for  $i = 1, \dots, \lceil c \cdot (25\ell/\lambda) \log n \rceil$  do
3:   sample an integer  $j \in \{0, 1, \dots, \lfloor \lambda/8 \rfloor\}$  uniformly at random
4:    $Y := \{v : (1 + \xi)^{j \cdot 8\ell/\lambda} \leq d(v) \leq (1 + \xi)^{(j+2) \cdot 8\ell/\lambda}\}$  ▷ compute  $Y$  only if  $\deg(Y) = O(m\ell/\lambda^2)$ 
5:   if  $\deg(Y) \leq 400m\ell/\lambda^2$  then
6:      $Z := Z \cup Y$ 
7: Propagate( $Z$ )
```

---

Note that in Randomized-Reset, the sum of degrees of the vertices in the sampled set  $Z$  is  $O(m\ell^2 \log n/\lambda^3)$ . Hence the total cost of Propagate( $Z$ ) that cannot be charged to estimate drops is  $O(m\ell^2 \log^2(n)/\lambda^3) = O(m \log^2(nW) \log^2(n)/(\lambda^3 \xi^2))$ .

The following lemma proves the correctness of Randomized-Reset.

Lemma 7.1. After Randomized-Reset( $\lambda$ ) completes, every  $v \in V$  is  $\lambda$ -certified w.h.p.

Proof. **TOPROVE 12** □

## References

- [ADV<sup>+</sup>25] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025*, pages 2005–2039. SIAM, 2025.
- [AHR<sup>+</sup>19] Bertie Ancona, Monika Henzinger, Liam Roditty, Virginia Vassilevska Williams, and Nicole Wein. Algorithms and hardness for diameter in dynamic graphs. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019*, volume 132 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [AIMN92] Giorgio Ausiello, Giuseppe F. Italiano, Alberto Marchetti-Spaccamela, and Umberto Nanni. On-line computation of minimal and maximal length paths. *Theor. Comput. Sci.*, 95(2):245–261, 1992.
- [AV14] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 434–443. IEEE Computer Society, 2014.
- [AvdB24] Anastasiia Alokina and Jan van den Brand. Fully dynamic shortest path reporting against an adaptive adversary. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024*, pages 3027–3039. SIAM, 2024.
- [AVW14] Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014*, volume 8572 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2014.
- [Ber09] Aaron Bernstein. Fully dynamic  $(2 + \epsilon)$  approximate all-pairs shortest paths with fast query and close to linear update time. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009*, pages 693–702. IEEE Computer Society, 2009.
- [Ber16] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. Comput.*, 45(2):548–574, 2016.
- [BGS20] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 1123–1134. IEEE, 2020.
- [BGS21] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental SSSP and approximate min-cost flow in almost-linear time. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 1000–1008. IEEE, 2021.
- [BGW20] Aaron Bernstein, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. Near-optimal decremental SSSP in dense weighted digraphs. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 1112–1122. IEEE, 2020.

- [BMN<sup>+</sup>04] Prosenjit Bose, Anil Maheshwari, Giri Narasimhan, Michiel H. M. Smid, and Norbert Zeh. Approximating geometric bottleneck shortest paths. *Comput. Geom.*, 29(3):233–249, 2004.
- [BT17] Arturs Backurs and Christos Tzamos. Improving viterbi is hard: Better runtimes imply faster clique algorithms. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 311–321. PMLR, 2017.
- [Che18] Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018*, pages 170–181. IEEE Computer Society, 2018.
- [CKL<sup>+</sup>24] Li Chen, Rasmus Kyng, Yang P. Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, s-t shortest path, and minimum-cost flow. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024*, pages 1165–1173. ACM, 2024.
- [CZ21] Shiri Chechik and Tianyi Zhang. Incremental single source shortest paths in sparse digraphs. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 2463–2477. SIAM, 2021.
- [CZ23] Shiri Chechik and Tianyi Zhang. Faster deterministic worst-case fully dynamic all-pairs shortest paths via decremental hop-restricted shortest paths. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*, pages 87–99. SIAM, 2023.
- [DFNdV24] Michal Dory, Sebastian Forster, Yasamin Nazari, and Tijn de Vos. New tradeoffs for decremental approximate all-pairs shortest paths. In *51st International Colloquium on Automata, Languages, and Programming, ICALP 2024*, volume 297 of *LIPIcs*, pages 58:1–58:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.
- [DI04] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
- [DP08] Ran Duan and Seth Pettie. Bounded-leg distance and reachability oracles. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008*, pages 436–445. SIAM, 2008.
- [DR18] Ran Duan and Hanlin Ren. Approximating all-pair bounded-leg shortest path and APSP-AF in truly-subcubic time. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 42:1–42:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
- [DSST89] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.
- [ES81] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981.

- [FGNS23] Sebastian Forster, Gramoz Goranci, Yasamin Nazari, and Antonis Skarlatos. Bootstrapping dynamic distance oracles. In 31st Annual European Symposium on Algorithms, ESA 2023, volume 274 of LIPIcs, pages 50:1–50:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
- [FNG23] Sebastian Forster, Yasamin Nazari, and Maximilian Probst Gutenberg. Deterministic incremental APSP with polylogarithmic update time and stretch. In Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, pages 1173–1186. ACM, 2023.
- [GVW20] Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New algorithms and hardness for incremental single-source shortest paths in directed graphs. In Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, pages 153–166. ACM, 2020.
- [HK95] Monika Rauch Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In 36th Annual Symposium on Foundations of Computer Science, FOCS 1995, pages 664–672. IEEE Computer Society, 1995.
- [HKN14] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In Symposium on Theory of Computing, STOC 2014, pages 674–683. ACM, 2014.
- [HKN15] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Improved algorithms for decremental single-source reachability on directed graphs. In Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, volume 9134 of Lecture Notes in Computer Science, pages 725–736. Springer, 2015.
- [HLS24] Bernhard Haeupler, Yaowei Long, and Thatchaphol Saranurak. Dynamic deterministic constant-approximate distance oracles with  $n^\epsilon$  worst-case update time. In 65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, pages 2033–2044. IEEE, 2024.
- [Ita86] Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(3):273–281, 1986.
- [JX22] Ce Jin and Yinzhan Xu. Tight dynamic problem lower bounds from generalized BMM and omv. In STOC ’22: 54th Annual ACM SIGACT Symposium on Theory of Computing, pages 1515–1528. ACM, 2022.
- [K19] Adam Karczmarz and Jakub cki. Reliable hubs for partially-dynamic all-pairs shortest paths in directed graphs. In 27th Annual European Symposium on Algorithms, ESA 2019, volume 144 of LIPIcs, pages 65:1–65:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [K20] Adam Karczmarz and Jakub cki. Simple label-correcting algorithms for partially dynamic approximate shortest paths in directed graphs. In 3rd Symposium on Simplicity in Algorithms, SOSA 2020, pages 106–120. SIAM, 2020.
- [KMG22] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. Incremental SSSP for sparse digraphs beyond the hopset barrier. In Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, pages 3452–3481. SIAM, 2022.

- [KMG24] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. A dynamic shortest paths toolbox: Low-congestion vertex sparsifiers and their applications. In Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, pages 1174–1183. ACM, 2024.
- [LVW18] Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, pages 1236–1252. SIAM, 2018.
- [Mad10] Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, pages 121–130. ACM, 2010.
- [Mao24] Xiao Mao. Fully dynamic all-pairs shortest paths: Likely optimal worst-case update time. In Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, pages 1141–1152. ACM, 2024.
- [MMN<sup>+</sup>25] Samuel McCauley, Benjamin Moseley, Aidin Niaparast, Helia Niaparast, and Shikha Singh. Incremental approximate single-source shortest paths with predictions, 2025. <https://arxiv.org/abs/2502.08125>.
- [RS11] Liam Roditty and Michael Segal. On bounded leg shortest paths problems. *Algorithmica*, 59(4):583–600, 2011.
- [RZ11] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.
- [ST14] Tong-Wook Shinn and Tadao Takaoka. Combining all pairs shortest paths and all pairs bottleneck paths problems. In LATIN 2014: Theoretical Informatics - 11th Latin American Symposium, volume 8392 of Lecture Notes in Computer Science, pages 226–237. Springer, 2014.
- [SVXY25] Barna Saha, Virginia Vassilevska Williams, Yinzhan Xu, and Christopher Ye. Fine-grained optimality of partially dynamic shortest paths and more. In Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025, pages 5147–5190. SIAM, 2025.
- [vdBN19] Jan van den Brand and Danupon Nanongkai. Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time. In 60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, pages 436–455. IEEE Computer Society, 2019.