Incremental
Approximate Single-Source Shortest Paths with Predictions

Samuel McCauley[*]
Williams College
sam@cs.williams.edu

Benjamin Moseley[†]
Carnegie Mellon University
moseleyb@andrew.cmu.edu

Aidin Niaparast[‡]
Carnegie Mellon University
aniapara@andrew.cmu.edu

Helia Niaparast[§]
Carnegie Mellon University
hniapara@andrew.cmu.edu

Shikha Singh[¶]
Williams College
shikha@cs.williams.edu

Abstract

The algorithms-with-predictions framework has been used extensively to develop online algorithms with improved beyond-worst-case competitive ratios. Recently, there is growing interest in leveraging predictions for designing data structures with improved beyond-worst-case running times. In this paper, we study the fundamental data structure problem of maintaining approximate shortest paths in incremental graphs in the algorithms-with-predictions model. Given a sequence σ of edges that are inserted one at a time, the goal is to maintain approximate shortest paths from the source to each vertex in the graph at each time step. Before any edges arrive, the data structure is given a prediction of the online edge sequence $\hat{\sigma}$ which is used to “warm start” its state.

As our main result, we design a learned algorithm that maintains $(1+\epsilon)$-approximate single-source shortest paths, which runs in $\tilde{O}(m\eta \log W/\epsilon)$ time, where W is the weight of the heaviest edge and η is the prediction error. We show these techniques immediately extend to the all-pairs shortest-path setting as well. Our algorithms are consistent (performing nearly as fast as the offline algorithm) when predictions are nearly perfect, have a smooth degradation in performance with respect to the prediction error and, in the worst case, match the best offline algorithm up to logarithmic factors. That is, the algorithms are “ideal” in the algorithms-with-predictions model.

As a building block, we study the *offline incremental* approximate single-source shortest-path (SSSP) problem. In the offline incremental SSSP problem, the edge sequence σ is known a priori and the goal is to construct a data structure that can efficiently return the length of the shortest paths in the intermediate graph G_t consisting of the first t edges, for all t. Note that the offline incremental problem is defined in the worst-case setting (without predictions) and is of independent interest.

1 Introduction

The efficiency of algorithms is typically measured in the worst-case model. The worst-case model makes a fundamental assumption that algorithmic problems are solved from scratch. In real applications, many problems are solved repeatedly on similar input instances. There has been a growing interest in improving

[*]Supported in part by NSF CCF 2103813

[†]Supported in part by Google Research Award, an Infor Research Award, a Carnegie Bosch Junior Faculty Chair, NSF grants CCF-2121744 and CCF-1845146.

[‡]Supported in part by the Air Force Office of Scientific Research under award number FA9550-23-1-0031 and NSF award CCF-1845146.

[§]Supported in part by NSF CCF-1845146.

[¶]Supported in part by NSF CCF 1947789

the running time of algorithms by leveraging similarities across problem instances. The goal is to *warm start* the algorithm; that is, initialize it with a machine-learned state, speeding up its running time. This initial state is learned from the past instances of the problem.

This recent line of work has given rise to a widely-applicable model for beyond-worst-case running time analysis. The area has come to be known as *algorithms with predictions*. In the algorithms-with-predictions model, the goal is to establish strong worst-case-style guarantees for the algorithm. The critical difference is that the running time of the algorithm is *parameterized* by the quality of the learned starting state; that is, the prediction quality. The algorithms designed in this model are also frequently referred to as *learning-augmented* or simply *learned* algorithms.

Ideally, an algorithm outperforms the best worst-case algorithm with high quality predictions (i.e. the algorithm is *consistent*). When the predictions are incorrect, the algorithm's performance should degrade proportionally to the error (i.e. the algorithm should be *smooth*) and never be worse than the best worst-case algorithm (i.e. the algorithm is *robust*). If an algorithm is consistent, smooth, and robust, it is called an *ideal learned algorithm*.

Kraska et al. [26] *empirically* demonstrated how machine-learned predictions can speed up data structures. This seminal paper inspired Dinitz et al. [15] to propose a *theoretical* framework to leverage predictions to speed up *offline* algorithms. Since this work, several papers have used predictions to improve the running time of offline combinatorial optimization problems such as maximum flow [12, 38], shortest path [28], and convex optimization [41]. Recently, Srinivas and Blum [43] give a framework for utilizing multiple offline predictions to improve the running time of online optimization problems. This growing body of theoretical and empirical work shows the incredible potential of using machine-learned predictions to improve the efficiency of algorithms. The prediction framework provides a rich and algorithmically interesting landscape that is yet to be understood.

Data structures are a critical building block for dynamic algorithms for optimization problems. There is a growing body of theoretical work on designing ideal algorithms for data structure problems [1, 2, 14, 30, 31, 34, 35, 45, 46]; see Section 1.1 for details. Predictions have been particularly effective at speeding up data structures for dynamic graph problems, which typically incur polynomial update times in the worst case. In particular, McCauley et al. [35] use predictions to design faster data structures for incremental topological maintenance and cycle detection. Henzinger et al. [21] and van den Brand et al. [45] initiate the use of predictions for designing faster dynamic graph algorithms. In particular, van den Brand [45] solve the online matrix-vector multiplication with predictions and use it to obtain faster algorithms for several dynamic graph problems such as incremental all-pairs shortest paths, reachability, and triangle detection. Liu and Srinivas [31] give efficient black-box transformations from offline divide-and-conquer style algorithms to fully dynamic learned algorithms that are given predictions about the update sequence. These initial results demonstrate incredible potential and there is a need to develop algorithmic techniques for designing data structures that can leverage predictions. Towards this goal, in this paper, we study the fundamental data structure problem of maintaining approximate single-source shortest paths in dynamic graphs with edge insertions. No learned data structure has been developed for this problem despite being a clear target in the area.

Incremental Single-Source Shortest Paths. In this paper, we design a data structure to maintain shortest paths in a weighted directed graph when edges are inserted over time. Initially, all nodes V of the graph are available and, in the single-source case, a source s is specified. There are m edges that arrive one by one: at each time step t, an edge (with a positive weight) is added to the graph. The goal is to design a data structure that approximately stores the shortest path distance from the source s to every other vertex v in the graph. Let $d^t(s, v)$ be the distance between s and v after t edge insertions. Let $\hat{d}^t(s, v)$ be an approximation of $d^t(s, v)$ computed by the algorithm after t edges arrive. The algorithm needs to efficiently compute $\hat{d}^t(s, v)$ such that $d^t(s, v) \leq \hat{d}^t(s, v) \leq (1 + \epsilon)d^t(s, v)$ for some constant $\epsilon > 0$.

Incremental shortest paths is a fundamental algorithmic problem used in applications as well as a building block for other data structures [20, 40]. This problem has been extensively studied in the literature [6, 7, 9, 17–19, 27]. The best-known worst-case update times for the problem are

$\tilde{O}(n^2 \log W/\epsilon^{2.5} + m)$ [39] for dense graphs and a $\tilde{O}(m^{4/3} \log W/\epsilon^2)$ algorithm [27] for sparse graphs.[1]

Recently, Henzinger et al. [21] and van den Brand [45] applied predictions to the problem of computing all-pairs shortest paths (APSP) in incremental graphs. We follow their prediction model in which the data structure is given a prediction of the online edge sequence $\hat{\sigma}$ before any edges arrive. The performance of the algorithm is given in terms of the prediction error. Define an edge e's error η_e as the difference between its arrival time in $\hat{\sigma}$ and σ and the aggregate error η as the $\max_e \eta_e$. They show that for the incremental APSP problem, predictions can be used to support $O(1)$-lookup time and $O(\eta^2)$ time per edge insert, which is optimal under the Online Matrix-Vector Multiplication Hypothesis [45]. The preprocessing time used in [21] is $O(mn^3)$, and it is $O(n^{(3+\omega)/2})$ in [45] where ω is the exponent from matrix multiplication.

Their work leaves open an important question—can predictions also help speed up the related and fundamental problem of maintaining approximate *single-source* shortest paths under edge inserts, which has not yet been studied in this framework.

Our Contributions. The main contribution of this paper is a new learned data structure for the incremental approximate SSSP problem and a demonstration that it is ideal (consistent, robust, and smooth) with respect to the prediction error. As a building block, we study the *offline* version of the problem that has not previously been considered, which is of independent interest.

We show that these techniques extend to the all-pairs shortest-path problem as well.

Offline Incremental Single-Source Shortest Paths. We give a new algorithm for the *offline* version of the incremental shortest path problem. In the offline version of the problem, the sequence of arriving edges is given in advance where each edge is assigned a unique time. The goal is to maintain approximate distances $\hat{d}^t(s, v)$ for all v and all times t as efficiently as possible. That is, given a query (v, t), the data structure outputs the approximate shortest path from source s to vertex v in the graph with edges inserted up to time t. By reversing time, the incremental and decremental versions of this problem are immediately equivalent.

Surprisingly, to our knowledge, past work in the offline setting has focused solely on exact, rather than approximate, incremental shortest path. These exact versions have strong lower bounds. Roddity and Zwick [40] show that for the incremental/decremental single-source shortest paths problem in weighted directed (or undirected) graphs, the amortized query/update time must be $n^{1-o(1)}$, unless APSP can be solved in truly subcubic time (i.e. $n^{3-\epsilon}$ for constant $\epsilon > 0$).

We show that the offline *approximate* version of the problem can be solved significantly faster than the exact version in the worst-case setting. This natural problem reveals key algorithmic ideas for designing our learned online SSSP algorithm.

Theorem 1. *For the offline incremental SSSP problem there exists an algorithm running in worst-case total time $O(m \log(nW)(\log^3 n)(\log \log n)/\epsilon)$ that returns $(1 + \epsilon)$ approximate single-source shortest paths for each time t.*

Predictions Model and Learned Online Single Source Shortest Paths. Let $\sigma = e_1, \ldots, e_m$ denote the actual online sequence of edge inserts. Before any edges arrive, the algorithm receives a prediction of this sequence, $\hat{\sigma}$. This is the same prediction model considered by Henzinger et al. [21] and van den Brand [45] for the all-pairs shortest-paths problem. Let $\text{index}(e)$ be the index of e in σ and $\widehat{\text{index}}(e)$ be the index of e in $\hat{\sigma}$. Define $\widehat{\text{index}}(e) := m + 1$ for edges e that are not in $\hat{\sigma}$. Let $\eta_e = |\text{index}(e) - \widehat{\text{index}}(e)|$ for each edge e in σ.

We first describe the performance of our learned SSSP algorithm in terms of parameters τ and $\text{HIGH}(\tau)$. For any τ, define $\text{HIGH}(\tau)$ to be the set of edges e in σ with error $\eta_e > \tau$. Then, we show that the bounds obtained are more robust than several natural measures of prediction error.

Theorem 2. *There is a learned online single-source shortest path algorithm that given a prediction $\hat{\sigma}$ gives the following guarantees:*

[1]The $\tilde{O}$ notation suppresses log factors.

- *The algorithm maintains a* $(1+\epsilon)$*-approximate shortest path among edges that have arrived.*
- *The total running time for all edge inserts is* $\tilde{O}(m \cdot \min_\tau\{\tau + |\mathit{HIGH}(\tau)|\} \log W/\epsilon)$.

Our algorithm uses the offline algorithm as a black box, and therefore our results also apply to the decremental problem in which edges are deleted one by one.

This theorem can be used to give results for two natural error measures. We call the first error measure, the *edit distance* $\text{Edit}(\sigma, \hat{\sigma})$, defined as the minimum number of insertions and deletions needed to transform σ to the prediction $\hat{\sigma}$. To the best of our knowledge this is a new error measure.

The second error measure is $\eta = \max_{e\in\sigma} \eta_e$ the maximum error of any edge; this measure was also used by past work (e.g. [34,35,45]). The theorem gives the following corollary.

Corollary 1. *There is a learned online algorithm that maintains* $(1+\epsilon)$*-approximate shortest paths and has running time at most the minimum of* $\tilde{O}(m \cdot \mathit{Edit}(\sigma, \hat{\sigma}) \log W/\epsilon)$ *and* $\tilde{O}(m\eta \log W/\epsilon)$.

The corollary can be seen as follows. By definition, there are no edges in $\text{HIGH}(\eta)$. Thus, setting $\tau = \eta$ in Theorem 2 gives an $\tilde{O}(m\eta \log W/\epsilon)$ running time. Alternatively, setting $\tau = \text{Edit}(\sigma, \hat{\sigma})$ ensures that $\text{HIGH}(\tau)$ contains at most τ edges. This is because any edge not inserted or deleted in the process of transforming $\hat{\sigma}$ to σ can move at most τ positions and so only inserted or deleted edges contribute to $\text{HIGH}(\tau)$. This gives a running time of $\tilde{O}(m \,\text{Edit}(\sigma, \hat{\sigma}) \log W/\epsilon)$.

Discussion On Single-Source Shortest Paths. Notice that if a small number of edges have a large error η_e, the $\text{Edit}(\sigma, \hat{\sigma})$ is small even though the maximum error is large, and thus the algorithm retains strong running time guarantees on such inputs. Furthermore, $\text{Edit}(\sigma, \hat{\sigma})$ is small even if there is a small number of edges that are not predicted to arrive but do, or are predicted to arrive but never do (such edges are essentially insertions and deletions).

On the other hand, the edit distance bound is a loose upper bound in some cases: for example, even if a large number of edges are incorrect, but have small relative change in position between σ and $\hat{\sigma}$, then η will be small.

The algorithm is *consistent* as its running time is optimal (since $\Omega(m)$ is required to read the input) up to log factors when the predictions are perfect. It is *smooth* in that it has a slow linear degradation of running time in terms of the prediction error. We remark that *robustness* to arbitrarily erroneous predictions can be achieved with respect to any worst-case algorithm simply by switching to the worst-case algorithm in the event that the learned algorithm's run time grows larger than the worst-case guarantee.

Extension to All-Pairs Shortest Paths. Next, we show that our techniques are generalizable by applying them to the all-pairs shortest-path (APSP) problem. Similar to the SSSP case, we first solve the offline incremental version of the problem by running the SSSP algorithm multiple times. We then extend it to the online setting with predictions.

For the incremental APSP problem, it does not make sense to consider amortized cost—Bernstein [5] gives an algorithm with nearly-optimal total work for the online problem even without predictions. As a result, past work on approximate all-pairs shortest path with predictions has focused on improving the worst-case time for each update and query.

For the APSP problem, we follow [21,45] and assume that $\hat{\sigma}$ is a permutation of σ—the set of edges predicted to arrive are exactly the set that truly arrive.[2]

Theorem 3. *There is a learned online all-pairs shortest path algorithm with* $\tilde{O}(nm \log W/\epsilon)$ *preprocessing time,* $O(\log n)$ *worst-case update time, and* $O(\eta^2 \log\log_{1+\epsilon}(nW))$ *worst-case query time.*

Comparison to Prior Work. To the best of our knowledge, no prior work has considered the incremental SSSP problem in the algorithms-with-predictions framework. Our algorithm for the SSSP problem has a recursive tree of subproblems, that we bring online with predictions on the input sequence. We

[2]While this is in some cases a strong assumption, it seems unavoidable for worst-case update and query cost.

remark that the work of Liu and Srinivas [31] gave a general framework for taking offline recursive tree algorithms into the online setting with predictions. Their framework is general and applies to a large class of problems that can be decomposed into recursive subproblems with smaller cost. In contrast, our tree-based decomposition technique is tailored specifically to shortest paths which enables our efficient runtime. Moreover, our analysis differs significantly from [31] as well—we cannot spread the cost evenly over smaller-cost recursive subproblems. This is because a single edge insert can result in $\Omega(n)$ changes in shortest paths. To avoid this, we allow subproblems to grow and shrink as necessary and charge the cost of a large subproblem to a large number of distance changes; see Section 3.

The incremental APSP problem with predictions was studied by past work [21,45]. Henzinger et al. [21] achieve $\tilde{O}(1)$ update time and $\tilde{O}(\eta^2)$ query time, after $O(mn^3)$ preprocessing for any weighted graph. van den Brand et al. [45] achieve similar update and query times with $\tilde{O}(n^{(3+\omega)/2})$ preprocessing time; however, the result is limited to unweighted graphs. They further show that this query time is essentially optimal: under the Online Matrix-Vector Multiplication Hypothesis, it is not possible to obtain $O(\eta^{2-\delta})$ query time for any $\delta > 0$ while maintaining polynomial preprocessing time and $O(n)$ update time. Thus, Theorem 3 obtains faster preprocessing time for sparse graphs and supports weighted graphs. Finally, we note that Liu and Srinivas [31] give bounds for the *fully dynamic* weighted APSP problem in the prediction-deletion model they propose and their bounds are incomparable to ours.

1.1 Additional Related Work

Data Structures with Predictions. Data structures augmented with predictions have demonstrated empirical success in key applications such as indexing [11,13,26], caching [25,32], Bloom filters [36,44], frequency estimation [23], page migration [24], routing [8].

Initial theoretical work on data structures with predictions focused on improving their space complexity or competitive ratio, e.g. learned filters [3,36,44] and count-min sketch [16,23] on stochastic learned distributions. Several papers have since used predictions to provably improve the running time of dictionary data structures, e.g. learned binary-search trees [10, 14, 29, 47]. McCauley et al. [34] presented the first "ideal" data structures with prediction for the list-labeling problem [34]. They use a "prediction-decomposition" framework to obtain their bounds and extend the technique to maintain topological ordering and perform cycle detection in incremental graphs with predictions [35].

Incremental SSSP and APSP in the Worst-Case Setting. We review the state-of-the-art deterministic worst-case algorithms for the incremental $(1+\epsilon)$ SSSP problem.

The best-known deterministic algorithm for dense graphs is by Gutenberg et al. [39] with total update time $\tilde{O}(n^2 \log W/\epsilon^{O(1)})$, which is nearly optimal for very dense graphs. For sparse graphs, Chechik and Zhang [9] give an algorithm with total time $\tilde{O}(m^{5/3} \log W/\epsilon)$, which is improved by Kyng et al. [27] to a total time $\tilde{O}(m^{3/2} \log W/\epsilon)$; this is the best-known deterministic algorithm for sparse graphs. Note that many of these solutions use the ES-tree data structure [42] as a key building block. The ES-tree can maintain exact distances in an SSSP tree for weighted graphs with total running time $O(mnW)$ [22], where W is the ratio of the largest to smallest edge weight. The ES-tree can be used to maintain $(1+\epsilon)$-approximate distances in total update time $\tilde{O}(mn \log W/\epsilon)$ for incremental/decremental directed SSSP; see e.g. [4,5,33].

This paper shows that even modestly accurate predictions can be leveraged to circumvent these high (polynomial) worst-case update costs in incremental SSSP.

For the approximate incremental APSP problem without predictions, Bernstein [5] gave an algorithm that has nearly-optimal total runtime $O(nm \log^4(n) \log(nW)/\epsilon)$ and $O(1)$ look-up time.

Peng and Rubinstein consider generally how incremental or decremental algorithms can be turned into fully dynamic algorithms [37].

2 Preliminaries

Let $\sigma = e_1, \ldots, e_m$ be the sequence of all edge insertions, and let V be the set of vertices. We assume m is of the form 2^k throughout this paper, which can be assumed without loss of generality by allowing dummy edge inserts. Each edge e_i has a weight $w(e_i) \in [1, W]$. Let G_t be the graph consisting of vertices V and the first t edges $e_1, \ldots, e_t$; we call this the *graph at time* t. We define G_0 to be the empty graph on the vertex set V.

The *length* of a path is the sum of the weights of its edges. Let $d^t(v)$ be the length of the shortest path from s to v in G_t. Throughout this paper, all logarithms are base 2 unless otherwise specified.

Problem Definition. In the offline problem, the algorithm is given a sequence of edge inserts $\sigma = e_1, e_2, \ldots, e_m$, the set of vertices V, and the source vertex s. The goal of the algorithm is to output a data structure that answers queries (v, t) of the form: what is the length of the shortest path from s to v at time t? This answer should be a $(1+\epsilon)$-approximation: specifically, for a query (v, t), if the data structure returns d, then $d^t(v) \leq d \leq d^t(v)(1+\epsilon)$.

In the online problem, the edges in $\sigma = e_1, \ldots, e_m$ arrive one at a time. Before any edge arrives, the algorithm is given a prediction $\hat{\sigma}$ of σ, as well as the set of vertices $V = \{v_1, \ldots, v_n\}$ and the source vertex s. We assume the length of $\hat{\sigma}$ is the same as σ. At all times, the algorithm must maintain an array D containing the length of the shortest path to each vertex. In particular, after edge t is inserted, D must satisfy $d^t(v_i) \leq D[i] \leq d^t(v_i)(1+\epsilon)$ for all vertices v_i.

Adjusting ϵ. Throughout this paper, we assume $\epsilon = O(1)$. In our algorithms, distance estimates $\hat{d^t}(v)$ are used for each node v and time t that satisfy the following invariant (see Lemmas 1 and 4): $d^t(v) \leq \hat{d^t}(v) \leq d^t(v)(1+\epsilon/\log m)^{\log m}$.

We will need to slightly adjust ϵ in our algorithms to account for lower-order terms. For $\epsilon < 1.79$, it is known that

$$\left(1 + \frac{\epsilon}{\log m}\right)^{\log m} \leq e^{\epsilon} \leq 1 + \epsilon + \epsilon^2,$$

which ensures that $d^t(v) \leq \hat{d^t}(v) \leq d^t(v)(1+\epsilon+\epsilon^2)$. To ensure our algorithms return $(1+\epsilon)$ approximations for $d^t(v)$, the ϵ parameters used are set to be $(\min\{1.79, \epsilon\})/4$.

Defining Recursive Subproblems. The offline algorithm is recursive. It divides the interval $[0, m]$ in half and recurses on both sides. On each recursive call, the algorithm will process an interval $[\ell, r]$ and further subdivide the interval until it contains a single edge. We call each recursive call $[\ell, r]$ a *subproblem*. We refer to this subproblem both as *subproblem* $[\ell, r]$ and *subproblem* x, where $x = (\ell + r)/2$ is the midpoint of $[\ell, r]$. Each subproblem corresponds to a node in the recursion tree. We might refer to the subproblem $[\ell, r]$ in the recursion tree as *node* x in the recursion tree. Each time $x \in \{1, \ldots, m-1\}$ is the midpoint of exactly one node in the tree. Let the *level* of a time x be the depth of node x in the recursion tree. So the level of $m/2$ is 1, the level of $m/4$ and $3m/4$ is 2, and so on. In particular, the level of x is one more than the maximum level of ℓ and r. For ease of notation, we set the level of 0 and m to be 0. See Figure 1 for an illustration of the recursion tree. We refer to the ancestors and descendants of a node in the recursion tree as *ancestors* and *descendants* of the corresponding subproblem. All these definitions extend to the online algorithm as well, as it is built up on the offline algorithm.

3 Technical Overview

3.1 Offline Algorithm

The preliminary observation behind our algorithm is the following. Let's say we round the distance to each vertex up to the nearest power of $(1+\epsilon)$. Since the maximum distance to any vertex is nW, this

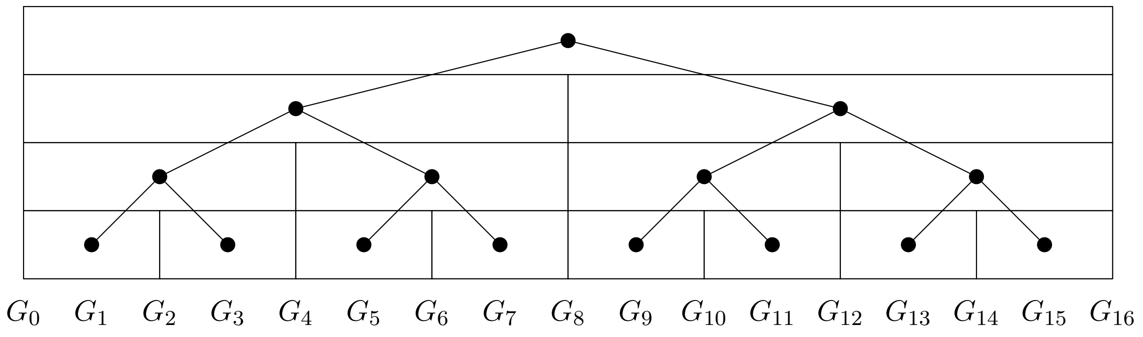
G_0 G_1 G_2 G_3 G_4 G_5 G_6 G_7 G_8 G_9 G_{10} G_{11} G_{12} G_{13} G_{14} G_{15} G_{16}

Figure 1: The recursion tree of the algorithm. Each node x in the tree is associated with an interval $[\ell, r]$ such that $x = (\ell + r)/2$. The depth of a node is the number of nodes in the path from the root $m/2$ to that node. For example, the depth of node $x = 6$ is 3, and its corresponding interval is $[4, 8]$.

means that we group distances into $O(\log_{1+\epsilon}(nW)) = O(\log(nW)/\epsilon)$ "buckets." Knowing that the distance to any vertex only decreases over time, the buckets of all vertices only change $O(n\log(nW)/\epsilon)$ times in total. The goal of our algorithm is to list the $O(\log(nW)/\epsilon)$ times when a vertex shifts from one bucket to the next.

Thus, when an edge is inserted at time t, our goal is to run Dijkstra's only on vertices whose distance is changing at time t, charging the cost to the changing distances.

Removing Vertices from Recursive Calls. Our algorithm recursively divides the sequence of edge inserts in half, starting with the entire sequence of edge inserts $\{e_1, \ldots, e_m\}$, and recursing until the sequence contains a single edge.

Consider a sequence $\{e_\ell, \ldots, e_r\}$, which we denote by $[\ell, r]$. We divide this interval into two equal halves: $[\ell, x]$, $[x, r]$.

The observation behind our algorithm is the following. Let's say we calculate the distance to all vertices at time x. Consider a vertex v that is in the same bucket at ℓ and x. Since the distance to v is non-increasing as new edges arrive, v must be in this bucket throughout the interval, and we do not need to continue calculating its distance. Therefore, we should remove v from the recursive call for the interval $[\ell, x]$. Similarly, if v is in the same bucket at x and r, then we should remove v from $[x, r]$.

If we are successful in removing vertices this way, we immediately improve the running time. Each vertex v is included in an interval $[\ell, r]$ if and only if it shifts from one bucket to another between ℓ and r. Thus, each time v shifts buckets, it is included in at most $\log m$ intervals. This means that each vertex is included in $O(\log m \log(nW)/\epsilon)$ intervals.

Let's assume that for an interval $[\ell, r]$, we can run Dijkstra's algorithm for the midpoint x in time (ignoring log factors) proportional to the number of edges that have at least one endpoint included in the interval $[\ell, r]$. Since each vertex contributes to the running time of $\text{polylog}(nmW)/\epsilon$ different subproblems, we can sum to obtain $\tilde{O}(m \log W/\epsilon)$ total cost to calculate the shortest path to every vertex in every subproblem.

Ensuring Dijkstra's Runs are Efficient. Let us discuss in more detail how to "remove" a vertex from a recursive call. We do not remove vertices from the graph; instead, we say that a vertex is *dead* during an interval $[\ell, r]$ if its distance bucket does not change during the interval, and *alive* otherwise. Each edge (u, v) is alive if and only if v is alive.

For each time x which is the midpoint of an interval $[\ell, r]$, consider building a graph G'_x consisting of all alive vertices and edges in $[\ell, r]$. If (u, v) is alive, but u is dead, we also add u to G'_x. Now, the time required to run Dijkstra's algorithm on G'_x is roughly proportional to the number of edges in G'_x as we desired, but the distances it gives are not meaningful—in fact, s may not even be in G'_x.

To solve this problem, we observe that we already know (approximately) the distance to any dead vertex u: we marked u as dead because its distance bucket does not change from ℓ to r. Thus, we add s to G'_x; then, for each alive edge (u, v) where u is dead, instead of adding u to G'_x, we add an edge from

s to v with weight equal to the rounded distance from s to u from the previous recursive call plus $w(u, v)$. Running Dijkstra's algorithm on G'_x now gives (ostensibly) meaningful distances from s.

We calculate the distance to each vertex in G'_x, round the distance up to obtain its bucket, and then recurse (marking vertices dead as appropriate) on $[\ell, x]$ and $[x, r]$.

Handling Error Accumulations. Unfortunately, the above method does not quite work, as the error accumulates during each recursive call.

For an interval $[\ell, r]$ with midpoint x, we use the terms *recursive call* $[\ell, r]$ and *recursive call* x interchangeably. For a recursive call x, the shortest path to v may begin with an edge (s, v') (for some dead vertex v') weighted according to the bucket of v'. The bucket of v' was calculated by rounding up the length of the shortest path to v' in some graph G'_y at time y; the length of this path again was also rounded up, and so on.

In other words, since the weight of each edge from s to a dead vertex v' is based on the bucket of v', it has been rounded up to the nearest power of $1 + \epsilon$. Thus, *each* recursive call loses a $1 + \epsilon$ approximation factor in distance.

To solve this, we note that the recursion depth of our algorithm is $\log_2 m$. Thus, we split into finer buckets: rather than rounding up to the nearest power of $1 + \epsilon$, we round up to the nearest power of $1+\epsilon/\log_2 m$. Thus, the total error accumulated over all recursive calls is $(1+\epsilon/\log_2 m)^{\log_2 m} \leq 1+\epsilon+\epsilon^2$ (see Section 2); decreasing ϵ slightly gives error $1 + \epsilon$.

Rounding buckets up to the nearest power of $1 + \epsilon/\log m$ changes our running time analysis: most critically, each vertex now changes buckets $\log_{1+\epsilon/\log m} nW = \Theta(\log(nW) \log m/\epsilon)$ times.

3.2 Ideal Algorithm with Predictions

With perfect predictions, the algorithm described above runs in $\tilde{O}(m \log W/\epsilon)$ time: we can just run the algorithm on the predicted (in fact the actual) sequence $\hat{\sigma} = \sigma$ to obtain an approximate distance to every vertex at each point in time. We show how to carefully rebuild portions of the offline approach to obtain an ideal algorithm.

Warmup: Hamming Distance of Error. To start, let's give a simple algorithm that uses predictions that contain error. Let $Ham(\sigma, \hat{\sigma})$ be the Hamming distance between σ and $\hat{\sigma}$—in other words, the number of edges whose predicted arrival time was not exactly correct in $\hat{\sigma}$. We give an algorithm that runs in $\tilde{O}(m \cdot Ham(\sigma, \hat{\sigma}) \log W/\epsilon)$ time.

The main idea behind this algorithm is to *update* the sequence of predictions $\hat{\sigma}$ as edges come in. At time t, an edge e arrives; if e arrived at t in $\hat{\sigma}$, the algorithm does nothing—the predictions created by running the offline algorithm on $\hat{\sigma}$ took e arriving at t into account. If e did not arrive at t in $\hat{\sigma}$, then the algorithm creates a new $\hat{\sigma}$, replacing the edge arriving at t with e. The algorithm then reruns the offline algorithm on the new $\hat{\sigma}$ in $\tilde{O}(m \log W/\epsilon)$ time.

Since we run the offline algorithm (in $\tilde{O}(m \log W/\epsilon)$ time) each time an edge was predicted incorrectly, we immediately obtain $\tilde{O}(m \cdot Ham(\sigma, \hat{\sigma}) \log W/\epsilon)$ time.

Handling Nearby Edges More Effectively. Ideally, we would not rebuild from scratch each time an edge is predicted incorrectly—we would like the running time to be proportional to *how far* an edge's true arrival time is from its predicted arrival time.

Our final algorithm improves over the warmup Hamming distance algorithm in two ways. First, it updates the predicted sequence $\hat{\sigma}$ more carefully. Second, it only rebuilds parts of the recursive calls of the offline algorithm: specifically, only intervals that changed as $\hat{\sigma}$ was updated.

First, let's describe how to update $\hat{\sigma}$. As before, when an edge e arrives at time t, if e was predicted to arrive at t in $\hat{\sigma}$ the algorithm does nothing. If e was predicted to arrive at time t' in $\hat{\sigma}$, the algorithm modifies $\hat{\sigma}$ by inserting e at time t (shifting all edges after t down one slot), and deleting e at time t'

(shifting all edges after t' up one slot). If e is not predicted in $\hat{\sigma}$, the algorithm only inserts e at time t, shifting subsequent edges down one slot.

The only entries in $\hat{\sigma}$ that change are those between t and t' (inclusive). Therefore, we only need to recalculate G' for times between t and t'.

Finally, we update the distance array D. The algorithm greedily updates D during the above rebuilds to maintain the invariant that $D[i]$ stores the estimated distance $\hat{d}^t(v_i)$, which as discussed in Section 2, is a $(1+\epsilon)$ approximation for $d^t(v_i)$.

Analysis. For any edge e that appears at time $\text{index}(e)$ in σ and time $\widehat{\text{index}}(e)$ in $\hat{\sigma}$, we define $\eta_e = |\text{index}(e) - \widehat{\text{index}}(e)|$. If e does not appear in $\hat{\sigma}$ then $\eta_e = |m+1-\text{index}(e)|$. Let η be the maximum error: $\eta = \max_{e\in\sigma} \eta_e$.

We show that an edge e causes a rebuild of a graph G'_t only if t is between its predicted arrival time $\widehat{\text{index}}(e)$ and its actual arrival time $\text{index}(e)$. This means that each G'_t can only be rebuilt $O(\eta)$ times. Since the total time to build all G'_t is $\tilde{O}(m \log W/\epsilon)$, the total time to rebuild all G'_t η times is $\tilde{O}(m\eta \log W/\epsilon)$.

With a more careful analysis, we can get the best of the Hamming analysis and the max error analysis. For any τ, let $\text{HIGH}(\tau)$ be the set of edges with error more than τ. For all edges with error more than τ, we may (in the worst case) rebuild the entire interval $\{1,\ldots,m\}$, for total cost $\tilde{O}(m|\text{HIGH}(\tau)|\log W/\epsilon)$. For all edges with error at most τ, the total rebuild cost is, as above, $\tilde{O}(m\tau \log W/\epsilon)$. Thus, we obtain total cost $\tilde{O}\left(m(\log W/\epsilon)\cdot \min_\tau\{\tau + |\text{HIGH}(\tau)|\}\right)$.

4 Offline Incremental Weighted Directed Shortest Path

This section presents an algorithm for the offline problem which takes $O(m\log(nW)(\log^3 n)\log\log n/\epsilon)$ time to build, and $O(\log\log_{1+\epsilon}(nW))$ time to answer a query (v,t).

The algorithm maintains an estimate of the shortest path at all points in time. Let $\hat{d}^t(v)$ be the estimate of $d^t(v)$ obtained by the algorithm. The algorithm does not explicitly maintain $\hat{d}^t(v)$ values for each vertex v and each time t, because of the following simple observation: if for times ℓ and r, $\ell < r$, we have $\hat{d}^\ell(v) = \hat{d}^r(v)$, it means that from the algorithm's perspective, node v has the same distance from s in graphs G_ℓ and G_r. Since the distances of the nodes from s are non-increasing over time, the algorithm infers that $\hat{d}^t(v) = \hat{d}^\ell(v)$ for each $\ell \le t \le r$. Although in such case $\hat{d}^t(v)$ is not explicitly stored by the algorithm, we still use this notation to refer to $\hat{d}^\ell(v) = \hat{d}^r(v)$.

In each subproblem x with an interval $[\ell, r]$, where $x = (\ell+r)/2$, vertices and edges are marked by the algorithm as alive or dead. A vertex is *alive* in subproblem x if its estimated distances are not the same in subproblems ℓ and r, otherwise it is *dead*. In each subproblem x (i.e., node x in the recursion tree), the distance estimates $\hat{d}^x(v)$ for all alive nodes v are explicitly maintained in a balanced binary search tree. This allows us to access $\hat{d}^x(v)$ in $O(\log n)$ time for each alive node v in subproblem x.

Moreover, for each node v, the algorithm maintains a list L_v of length $\log_{1+\epsilon}(nW)$, representing the times when v's estimated distance from s moves from one integer power of $(1+\epsilon)$ to another. In particular, the ith entry for a vertex v, denoted by $L_v(i)$, represents the minimum t such that $\hat{d}^t(v) \le (1+\epsilon)^i$. This is the data structure that the algorithm outputs in order to answer queries (v,t). To answer a query (v,t), i.e., to obtain a $(1+\epsilon)$-approximation of $d^t(v)$, the algorithm performs a binary search on $L(v)$ to find an i such that $L_v(i) \le t < L_v(i-1)$, and then we return $d = (1+\epsilon)^i$. The time needed to obtain d is then $\log(\log_{1+\epsilon}(nW))$. Note that $\hat{d}^t(v) \le d \le (1+\epsilon)\hat{d}^t(v)$, which as discussed in Section 2, it means that d is within a $(1+\epsilon)$ factor of $d^t(v)$ for the original ϵ.

Now we are ready to define the algorithm.

4.1 The Offline Algorithm

The offline algorithm is recursive. It starts with the interval $[0,m]$, and it recursively divides the interval in half and continues on the two subintervals. Initially, the algorithm marks all vertices and edges as alive in G_m, and all vertices as alive in G_0. The algorithm runs Dijkstra's on G_m, and for each vertex v, it sets $\hat{d}^m(v) = d^m(v)$. The algorithm sets $\hat{d}^0(s) = 0$, and for all $v \in V \setminus \{s\}$, it sets $\hat{d}^0(v) = \infty$. The algorithm then begins the recursive process on the interval $[0, m]$. The algorithm stops recursing when the interval $[\ell, r]$ contains only one new edge; that is, when $r - \ell = 2$.

Consider when the algorithm is given an interval $[\ell, r]$ to process. On this recursive call (subproblem), the goal is to calculate the distance estimates $\hat{d}^x(v)$ for alive nodes v, where $x = (\ell + r)/2$ is the midpoint of $[\ell, r]$. Since the algorithm processes the subproblems in the recursion tree from top to bottom, it has already processed the subproblems ℓ and r, i.e., the distance estimates are calculated for the alive nodes in G_ℓ and G_r.

Just to repeat, a vertex v is alive in G_x if it is alive in G_ℓ and G_r, and its estimated distance is *not* the same in G_ℓ and G_r, i.e., $\hat{d}^\ell(v) \neq \hat{d}^r(v)$. If a vertex is not alive, it is dead. A directed edge $e = (u, v)$ in G_x is said to be *alive* if v is alive in G_x; otherwise, e is *dead*. After the description of the algorithm, we discuss how to efficiently maintain alive and dead vertices and edges. Although the algorithm does not store $\hat{d}^x(v)$ for the dead vertices v in the subproblem x, we still use the notation $\hat{d}^x(v)$ to refer to $\hat{d}^\ell(v) = \hat{d}^r(v)$.

Now, we can define how the algorithm calculates $\hat{d}^x(v)$ for all alive vertices v. The algorithm creates a *new graph* G'_x whose vertex set is only the alive vertices in G_x along with s. Then, for each alive edge $e \in G_x$ with $e = (u, v)$, there are two cases:

- If both u and v are alive, add e to G'_x.
- If only v is alive, add an edge from s to v with weight $\hat{d}^x(u) + w(u, v)$ to G'_x.

To compute $\hat{d}^x(u)$, where u is a dead vertex in G_x, the algorithm needs to find the first ancestor of the current subproblem in the recursion tree in which u is alive. To do this, the algorithm does a binary search on the ancestors of node x in the recursion tree, and for the first ancestor x' where u is alive in $G_{x'}$, it recovers the value of $\hat{d}^{x'}(u) = \hat{d}^x(u)$ using the binary search tree stored at node x'.

The algorithm then computes the distance from s to each vertex in G'_x using Dijkstra's algorithm. For any alive vertex v, the algorithm stores the length of the shortest path to v found by Dijkstra's algorithm rounded up to the nearest integer power of $(1 + \epsilon/\log m)$ as $\hat{d}^x(v)$.

Efficiently Maintaining Alive Edges. For any x, let m_x be the number of alive edges in G_x. The algorithm maintains a list of all alive edges at each time x.

We now describe how to find the alive vertices and edges in G_x, where x is the midpoint of $[\ell, r]$. Note that if an edge (u, v) is alive in G_x, then its head v must be alive in G_x, which in turn implies that v is alive in G_ℓ and G_r. Since (u, v) has arrived before time r, it is alive in G_r. Therefore the alive edges in G_x are a subset of the alive edges in G_r. To obtain the list of alive edges in G_x, the algorithm iterates over the alive edges in G_r in $O(m_r)$ time, and removes the edges that either arrived after time x, or whose head node has the same estimated distance in both G_ℓ and G_r. While doing this, the algorithm additionally updates if each vertex is alive or not in G_x in $O(m_r)$ time.

From this, the algorithm constructs G'_x in $O(m_r)$ time, and runs Dijkstra's algorithm in $O(m_r \log n)$ time.

Efficiently Maintaining L_v Lists of Time Indexed Distances. To obtain the $L_v(i)$ values, initially, all the entries of L_v are empty. At each time x, when we are calculating the distance estimate $\hat{d}^x(v)$ for an alive node v, we update L_v: suppose $(1+\epsilon)^{i-1} < \hat{d}^x(v) \leq (1+\epsilon)^i$. If $L_v(i)$ is empty or $x < L_v(i)$, we set $L_v(i)$ to be x. Otherwise, $L_v(i)$ remains unchanged. In the end, the algorithm processes each of the lists, and for any empty $L_v(i)$, the algorithm sets it to be equal to the last non-empty entry of L_v before $L_v(i)$.

4.2 Analysis of the Offline Algorithm

This section establishes the correctness and running time guarantees of the algorithm.

Lemma 1. *If x is at level i, then for any vertex v, $d^x(v) \leq \hat{d}^x(v) \leq d^x(v)(1 + \epsilon/\log m)^i$.*

Proof. TOPROVE 0 □

With the correctness in place, the following lemma completes the proof of Theorem 1 by bounding the running time of the offline algorithm.

Lemma 2. *The offline algorithm runs in time* $O(m\log(nW)(\log^3 n)(\log\log n)/\epsilon)$.

Proof. TOPROVE 1 □

5 Incremental Shortest Paths with Predictions

This section gives the algorithm and analysis for the online version of the problem with predictions. Recall that in this model the edges arrive online. The goal of the algorithm is to maintain an array D of length n. At any time t, $D[i]$ should contain a $(1+\epsilon)$-approximation of $d^t(v_i)$.

5.1 The Algorithm

This section describes the online algorithm. Before any edges arrive, the algorithm is given a prediction $\hat{\sigma}$ of the edge arrival sequence.

Updated Predictions. Our algorithm dynamically maintains the predicted sequence of edge inserts by changing $\hat{\sigma}$ online. Specifically, after the tth edge arrival, the algorithm will construct a new prediction which we refer to as $\hat{\sigma}_t$. Initially, $\hat{\sigma}_0 = \hat{\sigma}$. We call the sequence $\hat{\sigma}_t$ the *updated prediction*. Intuitively, the algorithm modifies the updated prediction after each edge arrival based on which edge actually arrives.

For each edge e, let $\widehat{\text{index}}_t(e)$ be the position of e in $\hat{\sigma}_t$. Let $\text{index}(e)$ denote the position of e in σ (i.e. the true time when e arrives). If $e \notin \hat{\sigma}_t$, then we define $\widehat{\text{index}}_t(e) := m+1$.

Our algorithm updates the sequence of edges to agree with all edges seen so far. In other words, at time t, the algorithm maintains that for all edges e that arrive by t in σ, $\widehat{\text{index}}_t(e) = \text{index}(e)$.

Maintaining Metadata from the Offline Algorithm. Since the online algorithm continuously updates the result of the offline approach, it stores information to help it navigate the result of the offline algorithm.

Recall that the offline algorithm stores, for each time t, the set of vertices and edges alive at time t, as well as a distance estimate $\hat{d}^t(v)$ for any vertex v alive at time t. The online algorithm maintains exactly the same information. We will see that the algorithm updates these estimates continuously as the updated prediction changes.

Algorithm Description. First, let us describe the preprocessing performed by the algorithm before any edges arrive. The algorithm sets $D[i] = \infty$ for all i, except for $D[1] = 0$, which represents the source. Then, the algorithm runs the offline algorithm from Section 4 on $\hat{\sigma}$. By running the offline algorithm, the algorithm will store $\hat{d}^t(v)$ for all times t and all nodes v alive at time t.

Now, let us describe how the algorithm runs after the tth edge is inserted. At each time t, the algorithm *rebuilds* a subset of all the subproblems. These rebuilds update the precomputed $\hat{d}^t(v)$. When a subproblem is rebuilt, all of its descendants are rebuilt as well. The rebuilding procedure is formally described below.

Let e_t be the edge that arrives at time t, and let $t' = \widehat{\text{index}}_{t-1}(e_t)$, i.e., t' is the predicted arrival time for e_t in the updated predictions immediately before it arrives. Note that since the first $t-1$ edges of σ and $\hat{\sigma}_{t-1}$ are the same, we always have $t' \geq t$ (assuming the edges in the input sequence are distinct).

First, we describe how the algorithm updates $\hat{\sigma}_{t-1}$ to get $\hat{\sigma}_t$.

- If $t = t'$, i.e., the position of e_t is predicted correctly at the time it is seen, then set $\hat{\sigma}_t := \hat{\sigma}_{t-1}$.

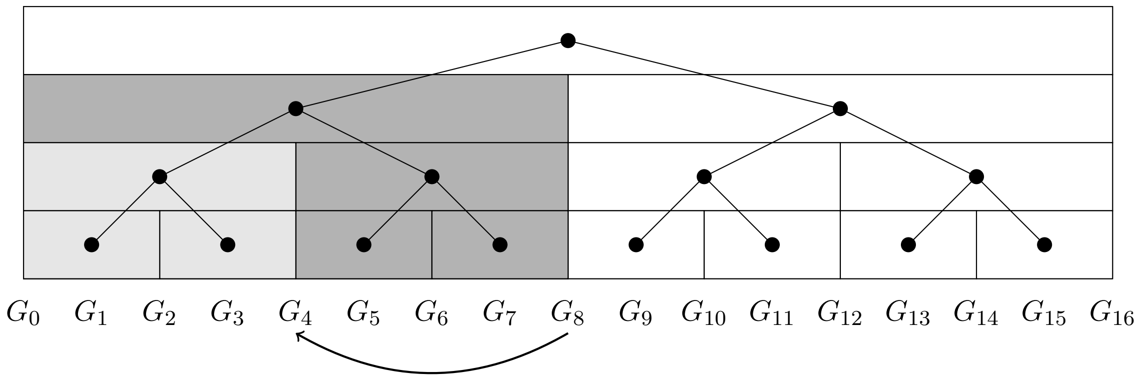
G_0 G_1 G_2 G_3 G_4 G_5 G_6 G_7 G_8 G_9 G_{10} G_{11} G_{12} G_{13} G_{14} G_{15} G_{16}

Figure 2: An illustration of the subproblems that get rebuilt during one edge insertion. In this example, at time $t = 4$, the edge e_4 was predicted to arrive but edge e_8 has arrived. So the algorithm moves edge e_8 from position $t' = 8$ to position $t = 4$. The algorithm then rebuilds all the subproblems with $t \leq (\ell + r)/2 < t'$ (colored dark gray) and their descendants (colored light gray) from top to bottom.

- If $t \neq t'$ and $t' \leq m$; that is, e_t is predicted to arrive at a later time. In this case, the algorithm moves e_t from position t' to position t in $\hat{\sigma}_{t-1}$, and shifts everything between t and t' one slot to the right to obtain $\hat{\sigma}_t$.
- If $t \neq t'$ and $t' = m + 1$; that is, at time $t - 1$, e_t is not in the predicted sequence. In this case, the algorithm inserts e_t in position t, shifts the rest of the sequence one slot to the right, and truncates the predicted sequence to length m to obtain $\hat{\sigma}_t$.

Rebuilding Subproblems. Next, the algorithm *rebuilds* subproblems. Recall that the algorithm given in Section 4 is recursive; each of its recursive calls can be represented by a node in the recursion tree.

To rebuild a subproblem $[\ell, r]$, the offline algorithm is called on $[\ell, r]$ using the updated prediction $\hat{\sigma}_t$. The rebuild makes recursive calls as normal. Any time a subproblem with midpoint x is rebuilt, the value of $\hat{d}^x(v)$ is updated based on the rebuild for all alive vertices v.

Let $[\ell_m, r_m]$ be the largest subproblem with $t \leq (\ell_m + r_m)/2 < t'$; then the algorithm rebuilds $[\ell_m, r_m]$. As mentioned above, all descendants of $[\ell_m, r_m]$ will be recursively called, and therefore rebuilt as well. In other words, the algorithm rebuilds all the subproblems $[\ell, r]$, with $t \leq (\ell + r)/2 < t'$, and all of their descendants from top to bottom (so in the first case, no subproblem gets rebuilt). See Figure 2 for an illustration. Let rebuild(t) be the set of all times t'' such that $t'' \in [\ell, r]$ for some $[\ell, r]$ rebuilt at time t. If no subproblem is rebuilt at time t, we define rebuild$(t) := \{t\}$ to insure that t is always in rebuild(t).

Updating the Distance Array. Finally, the algorithm must update the array D containing the estimated distance to each vertex. When a new edge is inserted, some of the entries in D need to be overwritten, as their estimated distance might have changed. At each time t, we want to have $D[i] = \hat{d}^t(v_i)$ for all i. To do so, the algorithm does the following for each time $t' \in$ rebuild(t), where $t' \leq t$, in sorted order. The algorithm iterates through all alive vertices v_i in $G_{t'}$, and sets $D[i] = \hat{d}^{t'}(v_i)$.

5.2 Analysis

This section establishes the theoretical guarantees of the algorithm. That is, the correctness of the approximation of the distances as well as the runtime bounds.

We begin with some structure that applies to any run of the *offline* algorithm. This will help us argue correctness of our algorithm.

Lemma 3. *For any sequence of edge inserts σ', let $\hat{d}^t(v)$ be the distance estimates that result from running the offline algorithm on σ'. Then for any vertex v and time $t \geq 1$, if v is dead at time t, then $\hat{d}^t(v) = \hat{d}^{t-1}(v)$.*

Proof. TOPROVE 2 □

If the edge that arrives at time t is predicted to arrive at time t' according to $\hat{\sigma}_{t-1}$, we say the edge *jumps* from t' to t. In such case, we say it *jumps over* all the positions $t \le i < t'$.

The next two lemmas prove the correctness of the algorithm.

Lemma 4. *For any time t and any vertex v, we have $d^t(v) \le \hat{d}^t(v) \le d^t(v)(1+\epsilon)$.*

Proof. TOPROVE 3 □

Lemma 5. *After inserting edge e_t, we have $D[i] = \hat{d}^t(v_i)$ for $i = 1, \dots, n$.*

Proof. TOPROVE 4 □

Now we determine the aggregate runtime of the online algorithm. Consider dividing σ into two sets of high- and low-error edges based on an integer parameter $\tau \ge 0$. Let $\text{LOW}(\tau) = \{e \in \sigma : |\text{index}(e) - \widehat{\text{index}}_0(e)| \le \tau\}$ and $\text{HIGH}(\tau) = \{e_1, \dots, e_m\} \setminus \text{LOW}(\tau)$. Therefore, $\text{HIGH}(\tau)$ is the set of edges whose initial predicted arrival time is more than τ slots away from their actual arrival time.

Lemma 6. *For any integer $\tau \ge 0$ and position $i \in \{1, \dots, m\}$, there are at most $\tau + 2|HIGH(\tau)|$ jumps over i.*

Proof. TOPROVE 5 □

We now prove the running time guarantees of the online algorithm.

Lemma 7. *The online algorithm runs in time $\tilde{O}\left(m \cdot \min_{\tau} \{\tau + |HIGH(\tau)|\} \cdot \log W/\epsilon\right)$.*

Proof. TOPROVE 6 □

Theorem 2 follows from Lemma 5 and the discussion in Section 2 for the approximation guarantees and Lemma 7 for the runtime.

6 All Pairs Shortest Paths

Our single-source approach can be run repeatedly to approximate distances between all pairs of vertices.

Specifically, the goal is to preprocess $G_0, \dots, G_m$ such that given i, j, and t, we can quickly find a $(1+\epsilon)$-approximation of $d^t(i,j)$, the distance from i to j in G_t. We run the single source shortest path algorithm for each source $s \in V$, storing a separate data structure for each. This requires $\tilde{O}(nm \log W/\epsilon)$ time and $\tilde{O}(n^2 \log W/\epsilon)$ space, gives the following corollary.

Corollary 2. *For the offline incremental all-pairs shortest-paths problem, there exists an algorithm running in total time $O(nm \log(nW) \log^3 n \log\log n/\epsilon)$ that returns $(1+\epsilon)$ approximate shortest paths for each pair of vertices for each time t.*

Online Learned APSP Algorithm. For the online setting, we consider the worst-case update and query bounds. In particular, the algorithm first preprocesses $\hat{\sigma}$. Then, it obtains the edges from σ one by one; the time to process any such edge is the update time. At any time during this sequence of inserts the algorithm can be queried for the distance between any two vertices in the graph; the time required to answer the query is the query time.

We can immediately combine this idea with the techniques of van den Brand et al. [45, Theorem 3.1] to obtain Theorem 3.

Proof. TOPROVE 7 □

7 Conclusion

Learned data structures have been shown to have strong empirical performance and have the potential to be widely used in systems. There is a need to develop an algorithmic foundation on how to leverage predictions to speed up worst-case data structures.

In this paper, we build on this recent line of work, and provide new algorithmic techniques to solve the fundamental problem of single-source shortest paths in incremental graphs. As our main result, we design an ideal (consistent, robust, and smooth) algorithm for this problem. Our algorithm is optimal (up to log factors) with perfect predictions and circumvents the high worst-case update cost of state-of-the-art solutions even under reasonably accurate predictions.

References

[1] Xingjian Bai and Christian Coester. Sorting with predictions. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[2] Ziyad Benomar and Christian Coester. Learning-augmented priority queues. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[3] Ioana O. Bercea, Jakob Bæk Tejs Houen, and Rasmus Pagh. Daisy Bloom Filters. In Hans L. Bodlaender, editor, *19th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2024)*, volume 294 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[4] Aaron Bernstein. Fully dynamic (2+ ε) approximate all-pairs shortest paths with fast query and close to linear update time. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 693–702. IEEE, 2009.

[5] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM Journal on Computing*, 45(2):548–574, 2016.

[6] 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, Denver, CO, USA, February 7-10, 2022*, pages 1000–1008. IEEE, 2021.

[7] Aaron Bernstein, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. Near-optimal decremental SSSP in dense weighted digraphs. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1112–1122. IEEE, 2020.

[8] Aditya Bhaskara, Sreenivas Gollapudi, Kostas Kollias, and Kamesh Munagala. Adaptive probing policies for shortest path routing. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:8682–8692, 2020.

[9] 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)*, pages 2463–2477. SIAM, 2021.

[10] Jingbang Chen and Li Chen. On the power of learning-augmented BSTs. *arXiv preprint arXiv:2211.09251*, 2022.

[11] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 155–171, 2020.

[12] Sami Davies, Benjamin Moseley, Sergei Vassilvitskii, and Yuyan Wang. Predictive flows for faster ford-fulkerson. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proc. of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 7231–7248. PMLR, 23–29 Jul 2023.

[13] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. ALEX: an updatable adaptive learned index. In *Proc. 46th Annual ACM International Conference on Management of Data (SIGMOD)*, pages 969–984, 2020.

[14] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, Aidin Niaparast, and Sergei Vassilvitskii. Binary search with distributional predictions. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[15] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Proc. 34th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 10393–10406, 2021.

[16] Elbert Du, Franklyn Wang, and Michael Mitzenmacher. Putting the "learning" into learning-augmented algorithms for frequency estimation. In *Proc. 38th Annual International Conference on Machine Learning (ICML)*, pages 2860–2869. PMLR, 2021.

[17] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Deterministic algorithms for decremental approximate shortest paths: Faster and simpler. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2522–2541. SIAM, 2020.

[18] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 674–683, 2014.

[19] 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, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I 42*, pages 725–736. Springer, 2015.

[20] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization. *SIAM J. Comput.*, 45(3):947–1006, 2016.

[21] Monika Henzinger, Barna Saha, Martin P. Seybold, and Christopher Ye. On the complexity of algorithms with predictions for dynamic graph problems. In Venkatesan Guruswami, editor, *15th Innovations in Theoretical Computer Science Conference, ITCS 2024, January 30 to February 2, 2024, Berkeley, CA, USA*, volume 287 of *LIPIcs*, pages 62:1–62:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[22] Monika Rauch Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 664–672. IEEE, 1995.

[23] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. Learning-based frequency estimation algorithms. In *Proc. 7th Annual International Conference on Learning Representations (ICLR)*, 2019.

[24] Piotr Indyk, Frederik Mallmann-Trenn, Slobodan Mitrovic, and Ronitt Rubinfeld. Online page migration with ML advice. In Gustau Camps-Valls, Francisco J. R. Ruiz, and Isabel Valera, editors, *International Conference on Artificial Intelligence and Statistics, (AISTATS)*, volume 151 of *Proceedings of Machine Learning Research*, pages 1655–1670. PMLR, 2022.

[25] Zhihao Jiang, Debmalya Panigrahi, and Kevin Sun. Online algorithms for weighted paging with predictions. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, (ICALP)*, volume 168 of *LIPIcs*, pages 69:1–69:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[26] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proc. 44th Annual International Conference on Management of Data, (SIGMOD)*, pages 489–504. ACM, 2018.

[27] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. Incremental sssp for sparse digraphs beyond the hopset barrier. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3452–3481. SIAM, 2022.

[28] Silvio Lattanzi, Ola Svensson, and Sergei Vassilvitskii. Speeding up bellman ford via minimum violation permutations. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18584–18598. PMLR, 2023.

[29] Honghao Lin, Tian Luo, and David Woodruff. Learning augmented binary search trees. In *Proc. 35th International Conference on Machine Learning (ICML)*, pages 13431–13440. PMLR, 2022.

[30] Honghao Lin, Tian Luo, and David P. Woodruff. Learning augmented binary search trees. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 13431–13440. PMLR, 2022.

[31] Quanquan C. Liu and Vaidehi Srinivas. The predicted-updates dynamic model: Offline, incremental, and decremental to fully dynamic transformations. In Shipra Agrawal and Aaron Roth, editors, *Proceedings of Thirty Seventh Conference on Learning Theory*, volume 247 of *Proceedings of Machine Learning Research*, pages 3582–3641. PMLR, 30 Jun–03 Jul 2024.

[32] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *Journal of the ACM (JACM)*, 68(4):1–25, 2021.

[33] Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 121–130, 2010.

[34] Samuel McCauley, Benjamin Moseley, Aidin Niaparast, and Shikha Singh. Online list labeling with predictions. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Proc. 36th Conference on Neural Information Processing Systems (NeurIPS)*, volume 36, pages 60278–60290. Curran Associates, Inc., 2023.

[35] Samuel Mccauley, Benjamin Moseley, Aidin Niaparast, and Shikha Singh. Incremental topological ordering and cycle detection with predictions. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 35240–35254. PMLR, 21–27 Jul 2024.

[36] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. *Proc. 31st Conference on Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[37] Binghui Peng and Aviad Rubinstein. Fully-dynamic-to-incremental reductions with known deletion order (e.g. sliding window). In Telikepalli Kavitha and Kurt Mehlhorn, editors, *2023 Symposium on Simplicity in Algorithms, SOSA 2023, Florence, Italy, January 23-25, 2023*, pages 261–271. SIAM, 2023.

[38] Adam Polak and Maksym Zub. Learning-augmented maximum flow. *Information Processing Letters*, 186:106487, 2024.

[39] 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*, pages 153–166, 2020.

[40] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

[41] Shinsaku Sakaue and Taihei Oki. Discrete-convex-analysis-based framework for warm-starting algorithms with predictions. In *35th Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[42] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981.

[43] Vaidehi Srinivas and Avrim Blum. Competitive strategies to use “warm start” algorithms with predictions. In *Proceedings of the 2025 ACM-SIAM Symposium on Discrete Algorithms, SODA 2025*. SIAM, 2025.

[44] Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. Partitioned learned bloom filters. In *Proc. 9th Annual International Conference on Learning Representations (ICLR)*, 2021.

[45] Jan van den Brand, Sebastian Forster, Yasamin Nazari, and Adam Polak. On dynamic graph algorithms with predictions. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 3534–3557. SIAM, 2024.

[46] Ali Zeynali, Shahin Kamali, and Mohammad Hajiesmaili. Robust learning-augmented dictionaries. In *Forty-first International Conference on Machine Learning*, 2024.

[47] Ali Zeynali, Shahin Kamali, and Mohammad Hajiesmaili. Robust learning-augmented dictionaries. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.