SIAM J. Comput.
Vol. 45, No. 2, pp. 548–574

MAINTAINING SHORTEST PATHS UNDER DELETIONS IN WEIGHTED DIRECTED GRAPHS[*]

AARON BERNSTEIN[†]

Abstract. We present an improved algorithm for maintaining all-pairs $(1+\epsilon)$ approximate shortest paths under deletions and weight-increases. The previous state of the art for this problem is *total* update time $\widetilde{O}(n^2\sqrt{m}/\epsilon)$ over all updates for directed unweighted graphs [S. Baswana, R. Hariharan, and S. Sen, *J. Algorithms*, 62 (2007), pp. 74–92], and $\widetilde{O}(mn/\epsilon)$ for undirected unweighted graphs [L. Roditty and U. Zwick, in *Proceedings of the* 45th *FOCS*, Rome, Italy, 2004, pp. 499–508]. Both algorithms are randomized and have constant query time. Very recently, Henzinger, Krinninger, and Nanongkai presented a deterministic version of the latter algorithm [M. Henzinger, S. Krinninger, and D. Nanongkai, in *IEEE FOCS*, 2013, pp. 538–547]. Note that $\widetilde{O}(mn)$ is a natural barrier because even with a $(1+\epsilon)$ approximation, there is no $o(mn)$ combinatorial algorithm for the *static* all-pairs shortest path problem. Our algorithm works on directed weighted graphs and has total (randomized) update time $\widetilde{O}(mn\log R/\epsilon)$ where R is the ratio of the largest edge weight to appear at any point in the update sequence to the smallest such weight. (As with previous algorithms, our query time is constant.) Technically, the running time is $\widetilde{O}(mn\log R/\epsilon) + O(\Delta)$, where Δ is the total number of updates; the same $O(\Delta)$ term is also implicitly present in all other algorithms for the problem, since a constant time per update is clearly unavoidable. Note that $\log R = O(\log(n))$ as long as weights are polynomial in n; thus, we effectively expand the $\widetilde{O}(mn/\epsilon)$ total update time bound from undirected unweighted graphs to directed graphs with polynomial weights. This is in fact the first nontrivial algorithm for decremental all-pairs shortest paths that works on weighted graphs (previous algorithms could only handle small integer weights). By a well-known reduction from decremental algorithms to fully dynamic ones [M. Henzinger and V. King, in *Proceedings of the* 36th *FOCS, Milwaukee, WI*, 1995, pp. 664–672], our improved decremental algorithm leads to improved query-update trade-offs for fully dynamic $(1+\epsilon)$ approximate all-pairs shortest paths (APSP) algorithms in directed graphs.

Key words. graph algorithms, dynamic algorithms, approximation, shortest paths

AMS subject classification. 05C85

DOI. 10.1137/130938670

1. Introduction.

Dynamic graphs are used to model settings where we need to maintain some information about a graph that is changing over time. We focus on the problem of maintaining shortest paths in a graph whose edges are being inserted and deleted. More formally, the objective of the dynamic single source shortest path problem (SSSP) is to efficiently process an online sequence of update and query operations. An update can insert or delete an edge, or change the weight of an existing edge. A query asks for the distance in the *current* graph from the source to a given vertex v. In dynamic all-pairs shortest paths (APSP), the query can ask for the distance between any pair vertices. A dynamic algorithm is said to be *incremental* if the updates are only insertions and weight-decreases, *decremental* if they are only deletions and weight-increases, and *fully dynamic* if all updates are allowed.

[*]Received by the editors September 26, 2013; accepted for publication (in revised form) July 1, 2015; published electronically April 27, 2016. This publication is a full length version of an extended abstract published in STOC 2013 [5]. This work was supported by the NSF graduate research fellowship (GRFP).
http://www.siam.org/journals/sicomp/45-2/93867.html

[†]Department of Computer Science, Columbia University, New York, NY, 10027 (bernstei@gmail.com).

1.1. Existing algorithms. The efficiency of dynamic shortest path algorithms is judged by two parameters: query time and update time. One can achieve many different trade-offs between these, but typically the goal is to minimize update time while keeping the query time polylogarithmic. The naive algorithm simply recomputes shortest paths from scratch after every update; the query time is O(1), while the update time is $\widetilde{O}(m)$ for SSSP and $\widetilde{O}(mn)$ for APSP.[1]

The classic dynamic shortest path problem is maintaining a shortest path tree under deletions. In 1981, Even and Shiloach [8] gave an algorithm with *total* update time $O(mn)$ in undirected unweighted graphs (amortized update time $O(n)$). King [12] later extended this work to directed graphs. Unfortunately, the results for the single source problem essentially stop here. Nothing nontrivial is known for weighted graphs, and nothing nontrivial is known for the fully dynamic case. Moreover, Roditty and Zwick showed a reduction from boolean matrix multiplication to decremental SSSP in undirected unweighted graphs, explaining the lack of progress for beating $\widetilde{O}(mn)$ total update time [16]. The only progress we know of is for the specific case of $(1+\epsilon)$ approximate decremental SSSP in unweighted, undirected graphs [6]: here we can achieve total update time $o(n^{2+\epsilon})$ for any fixed $\epsilon > 0$.

In contrast to dynamic SSSP, there are a large number of papers on dynamic APSP algorithms that beat the trivial $\widetilde{O}(mn)$ total update time, of which we describe just a few. For the fully dynamic case, a long string of results culminated in a breakthrough paper of Demetrescu and Italiano [7], which showed that in general graphs we can maintain constant query time while achieving $\widetilde{O}(n^2)$ amortized update time. Sankowski used matrix multiplication in unweighted graphs to achieve randomized update time $O(n^{1.932})$ and randomized query time $O(n^{1.288})$ [17]. If we allow a $(2+\epsilon)$-approximation, Bernstein showed in 2009 [3] that for undirected graphs we can achieve an $O(\log\log\log(n))$ query time with amortized update time $o(mn^{\epsilon}\log R/\epsilon)$ for any fixed $\epsilon > 0$; R is the ratio between the largest and the smallest edge weights.

In the decremental setting edges are never inserted into the graph, so for unweighted graphs at least, the number of updates is finite. As such, it is easier to analyze this setting not in terms of amortized update time but in terms of the *total* update time over all deletions (in unweighted graphs, this is just m times the amortized update time). The naive algorithm has total update time $\widetilde{O}(m^2 n)$. Baswana, Hariharan, and Sen [2] showed that in directed unweighted graphs we can decrementally maintain APSP with constant query time and *total* update time $\widetilde{O}(n^3)$. They showed in the same paper that if we allow a $(1+\epsilon)$ approximation, we can reduce the total update time to $\widetilde{O}(n^2\sqrt{m}/\epsilon)$. Roditty and Zwick [14] then showed that if we allow a $(1+\epsilon)$ approximation in *undirected*, unweighted graphs, we can reduce the total update time to $\widetilde{O}(mn/\epsilon)$.

All of these decremental algorithms are randomized; in particular, they only work on an oblivious adversary, which does not get to see the random choices made by the algorithm. This is true of our algorithm as well. Very recently, however, in a paper to appear in the *Proceedings of FOCS* 2013, Henzinger, Krinninger, and Nanongkai presented a *deterministic* decremental algorithm for $(1+\epsilon)$-approximate APSP that achieves basically the same results as Roditty and Zwick's randomized algorithm [14]: total update time $\widetilde{O}(mn/\epsilon)$ and query $O(\log\log(n))$ (not quite constant). This result only works on unweighted, undirected graphs [10].

In dense graphs, we can actually improve the total update time of $\widetilde{O}(mn)$ if we allow a slightly worse approximation. Bernstein and Roditty showed that in unweighted,

[1] We say that $f(n) = \widetilde{O}(g(n))$ when we have that $f(n) = \widetilde{O}(g(n)\text{polylog}(n))$.

undirected graphs we can decrementally maintain $(3+\epsilon)$-approximate APSP in total update time $\widetilde{O}(n^{2.5+1/\sqrt{\log(n)}})$. Very recently, in the same FOCS 2013 paper [10], Henzinger, Krinninger, and Nanongkai improved upon this bound. Their algorithm has a slightly smaller total update time of $\widetilde{O}(n^{2.5})$ and has a reduced approximation error of $(2+\epsilon)$. They can even reduce the multiplicative approximation error to $(1+\epsilon)$ by introducing an additive error of 2 (same update time). These algorithms also only work in unweighted, undirected graphs.

1.2. Our contributions. We present a new algorithm for decremental approximate APSP. Note that if we restrict ourselves to only a $(1+\epsilon)$ approximation, all the above results for this problem are tending toward a natural $\widetilde{O}(mn)$ barrier; even with a $(1+\epsilon)$ approximation, there is no $o(mn)$ algorithm for computing *static* APSP, so we have little hope of beating $O(mn)$ in the dynamic case (if we allow more than a $(1+\epsilon)$ approximation, then we can, in fact, beat $O(mn)$ in the static case, which explains the recent results beating it in the dynamic case). Thus, the Roditty and Zwick algorithm [14] and later the Henzinger, Krinninger, and Nanongkai algorithm [10] achieve the best update time we can hope for with a $(1+\epsilon)$ approximation. However, both these results only work in undirected unweighted graphs. This raises the question of whether we can reach $\widetilde{O}(mn)$ for a more general case. Directed graphs? Weighted graphs? Can we remove the $(1+\epsilon)$ approximation? We make some headway toward answering this question. In particular, we prove the following theorem.

THEOREM 1. *Let G be a directed graph with real positive weights subject to an online sequence of update and query operations, where a query operation asks for a $(1+\epsilon)$-shortest distance between any pair of vertices, while an update operation deletes an edge, or increases the weight of an edge. There exists an algorithm that has worst-case query time $O(1)$ (a single table look-up), and that processes all updates in* total *time $O(mn\log^4(n)\log(nR)/\epsilon+\Delta)$, where Δ is the total number of update operations, and R is the ratio of the heaviest edge weight to appear in G at any point in the update sequence to the lightest such edge weight. For unweighted graphs the running time is slightly smaller: $O(mn\log^4(n)\log\log(n))$. The update procedure is randomized (Monte Carlo) and assumes an oblivious adversary.*

Note that the $O(\Delta)$ factor in our total update time has nothing to do with the particularities of our algorithm but is simply an unavoidable constant time per update (no matter what we do, we cannot avoid looking at every update). The only reason $O(\Delta)$ did not come up in the other decremental algorithms mentioned above is because they only worked for unweighted graphs, in which we always have $\Delta \leq m$ because every update deletes an edge. But in weighted graphs, we can no longer bound the number of updates. It may thus appear strange that we continue to analyze our algorithm in terms of *total* update time, but the basic idea is that our algorithm in fact spends a total of only $\widetilde{O}(mn\log R/\epsilon)$ time processing updates that might actually be relevant (i.e., might actually change some approximate shortest distance); it is not hard to see that since distances only increase, any given $x-y$ distance can increase by a $(1+\epsilon)$ factor at most $\log_{(1+\epsilon)}(nR) = O(\log(nR)/\epsilon)$ times, so there are at most $O(n^2\log(nR)/\epsilon)$ relevant updates that require processing. (Note that although we do not know ahead of time which are relevant, this difficulty is not hard to deal with.) The additional $O(1)$ time per update then corresponds to merely throwing away the irrelevant updates.

Note that $\log(nR) = O(\log(n))$ as long as weights are polynomial in n. Thus, our algorithm achieves the desired $\widetilde{O}(mn/\epsilon)$ total update time for directed graphs with weights polynomial in n, as compared to the previous state of the art of Roditty and

Zwick [14], which only achieved this bound for undirected unweighted graphs. In fact, ours is the first nontrivial algorithm for decremental APSP in weighted graphs (though previous ones could handle small integer weights). Another advantage of our algorithm over the one of Roditty and Zwick is that although their query time was constant, it still required an involved process, whereas ours is simply a table look-up. One obvious benefit of this is that in real-world scenarios, one often wants to keep query time as small as possible. Another benefit is that in many settings, when an update occurs, we want to quickly find out all pairs that were affected by it. The involved query procedure of Roditty and Zwick would require them to separately check each pair, so that even if only a few pairs were affected by the update, the algorithm would still require a prohibitive $\widetilde{O}(n^2)$ time to find those pairs. Our algorithm, however, could just return all changed entries of our distance matrix; this requires time O (number of changed entries) and so does not increase the asymptotic update time.

There is a standard reduction from decremental APSP algorithms to fully dynamic APSP algorithms with a query-update trade-off. Thus, our improved decremental algorithm leads to a new fully dynamic one. For any $T \leq \sqrt{n}$, we can maintain $(1+\epsilon)$-APSP in directed graphs with amortized update time $\widetilde{O}(\frac{mn \log R}{T\epsilon})$ and query time $O(T)$. This trade-off was previously only possible for undirected unweighted graphs. The decremental to fully dynamic reduction was originally introduced by Henzinger and King [9] and has since been used in several other dynamic shortest paths papers [16], [14]. Our use of this reduction is more or less identical to previous ones, but a few of the details differ, so we offer a more detailed discussion at the end of the paper (section 8.3).

This paper presents our result as a decremental algorithm, but like many decremental algorithms, it works equally well in the incremental setting where you have only edge insertions and weight decreases. The algorithm for the two settings is essentially identical, but we picked the decremental setting because it tends to be the more difficult of the two and is arguably the more useful; for example, the above reduction to a fully dynamic algorithm with a query-update trade-off only works if one starts with a decremental algorithm. For the rest of the paper we focus only on the decremental setting and leave our discussion of the incremental setting for the very end (section 8.2).

Section 4 outlines the basic approach of our algorithm, but first we present notation and existing work in sections 2 and 3. Section 5 introduces a simpler version of our algorithm for the sake of intuition, and section 6 presents our final algorithm, which proves the main theorem above. Section 7 presents in full detail an existing result that we rely heavily upon as well as a new improvement on this result that reduces the algorithm's dependence on Δ. Finally, section 8 touches upon some final details, including applications of our algorithm to the incremental and fully dynamic setting.

2. Preliminaries.

Let $G = (V, E)$ be a directed graph with real positive weights. As we process our updates, G always refers to the *current* version of the graph. Let m be the number of edges in the *initial* graph, and let n be the number of vertices (which does not change); we assume that $m = \Omega(n)$. Given any vertices x, y, let (x, y) be the edge between them (if it exists), and let $w(x, y)$ be the weight of this edge. Let $\pi(x, y)$ be the shortest $x - y$ path in G (if one exists); if there are multiple shortest paths from x to y, we can use any tie breaking strategy which ensures that any subpath of a shortest path is itself a shortest path. (For an example, see section 3.4 of [7].) Define $\delta(x, y)$ to be the length of $\pi(x, y)$, or ∞ if no $x - y$ path exists. We assume that all edge weights are positive. We define the *hop-length* of a path P, denoted $h(P)$, to be the number of edges on P, and we let $h(x, y) = h(\pi(x, y))$. For any h, we define $\pi^h(x, y)$ to be the shortest path from x to y that uses at most h edges (if one exists), and we define $\delta^h(x, y)$ to be the length of $\pi^h(x, y)$, or ∞ if this path does not exist. Given a graph G' different from G, we define $\pi_{G'}(x, y), \delta_{G'}(x, y)$, $\pi^h_{G'}(x, y)$, and $\delta^h_{G'}(x, y)$ to be the corresponding paths and distances in G'.

Many of our running times are expressed in terms of the variables Δ and R. We define Δ to be the total number of updates made to the graph over the course of the entire dynamic sequence. We define R to be the ratio of the largest weight in the graph *at any point in the update sequence* to the smallest such weight. More formally, we define C, c, and R as follows:

- $C = \max_\tau \max_{(u,v)\in E}\{w(u, v) \text{ at time } \tau\}$,
- $c = \min_\tau \min_{(u,v)\in E}\{w(u, v) \text{ at time } \tau\}$,
- $R = C/c$.

Note that as long as weights are polynomial in n, $\log R = O(\log(n))$. Finally, we say that output $\delta'(x, y)$ is α-approximate if $\delta(x, y) \leq \delta'(x, y) \leq \alpha\delta(x, y)$. We say that a path $P(x, y)$ is α-approximate if its weight is an α-approximation of $\delta(x, y)$.

For the sake of simplicity, we make a few minor assumptions about the graph and the update sequence.

- We assume that R is known in advance; in section 8.1, we show that this assumption can easily be removed by continually updating an approximate guess for R.
- We model the deletion of an edge by increasing its weight to ∞. This is not quite rigorous, as then $\log(R)$ also becomes infinite. To resolve this, we model the deletion of an edge by raising its weight to large number U^*. We ensure that at all times $U^* \geq 2nC^*$, where C^* is the larger noninfinite weight in the current graph. Thus, if a query ever returns a shortest $x - y$ distance $\geq U^*$, this clearly corresponds to there being no path from x to y in the graph. As edge weights in the graph increase, U^* might come to be less than $2nC^*$, in which case we repeatedly double it until it is large enough. It is not hard to see that modeling deletions in this way does not affect the asymptotic running time. First, repeated doubling can never cause U^* to be greater than $4nC$, so adding edges of weight U^* increases R by only an $O(n)$ factor, so the $\log(nR)$ term is not affected. Second, U^* doubles at most $O(\log(4nC/c)) = O(\log(nR))$ times, so the dummy weight on each deleted edge increases at most $O(\log(nR))$ times, and the total number of additional updates is at most $O(m \log(nR))$; the change to $O(\Delta)$ is thus well within the $\widetilde{O}(mn \log(nR))$ total update time.
- We assume the graph is connected at all times. Our algorithm does not actually rely on this, but it obviates the need for an analysis of edge cases. We can ensure this by adding a super source s^* with an edge of weight U^* to and from every vertex; as above, U^* might increase as edge weights in the original graph increase. A shortest distance $\geq U^*$ once again indicates that no path exists. The number of edges is still $O(m)$, and the number of new updates is only $O(n \log(nR))$.

3. Maintaining a shortest path tree.

All existing decremental algorithms for APSP rely on a subroutine for maintaining a shortest path tree under deletions. The original result comes from Even and Shiloach [8], and it was later extended to directed weighted graphs by King [12]. We use the following from [12] (Ramalingam and Reps [13] prove a similar result).

THEOREM 2 (see [12]). *Given a source s, it is possible to fully dynamically maintain a shortest path tree from (or to) s in such a way that the update time for update σ is $O(1+E(\sigma))$, where $E(\sigma)$ is the number of edges incident to vertices whose distance from source s changed as a result of update σ. That is, we only touch the edges of a vertex when the distance to that vertex changes.*

COROLLARY 3.1. *Given a source s in a graph with positive integer weights, we can* decrementally *maintain a shortest path tree up to distance d—that is, a shortest path tree truncated at depth d—in total update time $O(md)$ over all deletions.*

Proof (of Corollary 3.1). All distances in the shortest path tree are between 1 and d, and the update sequence is decremental, so distances are only increasing. This implies that the distance to any particular v can change at most d times. We thus touch the incident edges of each vertex $O(d)$ times over all deletions, leading to a total update time of $O(md)$. ☐

In an earlier paper [3], we developed a simple but powerful generalization of the above corollary, though at the cost of a $(1+\epsilon)$ error. Loosely speaking, we showed that instead of maintaining a shortest path tree up to distance d, we can efficiently maintain it up to a hop-length h. We refer to this algorithm for decrementally maintaining approximate single source shortest distances as the h-SSSP algorithm. We now formally present the result achieved by h-SSSP, which we use a building block throughout most of the paper, leaving the details of h-SSSP itself for section 7.

THEOREM 3 (see [3]). *Given a source s and a hop distance h, h-SSSP decrementally maintains distances $\delta'(s,v)$ to each vertex v, such that we always have $\delta(s,v) \leq \delta'(s,v) \leq (1+\epsilon)\delta^h(s,v)$. Moreover, after every update h-SSSP can return a list of all vertices v for which $\delta'(s,v)$ changed due to that update, without affecting the asymptotic update time. The total update time of h-SSSP over all deletions and weight-increases is $O(mh\log(n)\log(nR)/\epsilon + \Delta)$ for weighted graphs and a slightly faster $O(mh\log(n)\log\log(n)/\epsilon)$ for unweighted ones.*

Remark. Note that in the theorem above $\delta'(s,v)$ itself may correspond to a path with more than h edges; the algorithm is not concerned with the length of the output path. The only guarantee is merely that $\delta'(s,v)$ is a good approximation to $\delta^h(s,v)$, which is equal to $\delta(s,v)$ as long as $h(s,v) \leq h$. Thus, we can think of our algorithm as maintaining $(1+\epsilon)$-distances from s up to hop-length h.

When we say that we "run" the h-SSSP algorithm from (to) vertex s, this refers not merely to an initialization step, but rather to the whole dynamic procedure. In other words, it means that we maintain approximate distances $\delta'(s,v)$ to (from) each vertex v over all deletions to come. By Theorem 3 the total cost of running the h-SSSP algorithm is $\widetilde{O}(mh\log R/\epsilon)$.

4. The basic approach.

The basic outline of our approach is similar to one Bernstein used in two earlier papers [3, 4], though except for the h-SSSP algorithm essentially all the details differ. The advantage of h-SSSP over King's $O(md)$ algorithm [12] (see Corollary 3.1) is that the latter maintains a shortest path tree up to a certain distance, whereas h-SSSP maintains it up to a certain hop-length, and is hence barely dependent on the weights of the edges. (The running time of h-SSSP does depend on $\log R$, but this is only a logarithmic dependence on the weight, as compared to King's linear dependence.) This change of focus from weighted distance to hop-length is obviously crucial for weighted graphs, but it is in fact equally important in unweighted graphs. Both d and h can initially be as large as $n-1$, but whereas the distance between a pair vertices is inherent to the graph, the *hop* distance can easily be decreased by adding shortcuts to the graph.

Suppose that we already knew the shortest distance $\delta(v, w)$. We could then add a new edge (v, w) of weight $\delta(v, w)$; this would not change any of the distances in the graph, but it would reduce $h(v, w)$ to one. It would also reduce the hop-length of any path that used $\pi(v, w)$ as a subpath. This observation suggests the following approach: we construct a large number of shortcut edges to reduce hop distances all across the graph, which would allow us to efficiently run the h-SSSP algorithm. The problem is that in order to create shortcut edges we need to have already computed $\delta(v, w)$; moreover, as the graph changes, so do the shortest distances in the graph, so dynamically maintaining correct weights on the shortcut edges requires maintaining the distances $\delta(v, w)$.This leaves us in the position of trying to maintain shortest paths by first maintaining other shortest paths.

Bernstein previously applied the idea of creating shortcut edges to reduce the hop-diameter in two papers on undirected graphs [3, 4]. (Note that these papers were not actually on the problem of decremental shortest paths; they just used the same basic approach of shortcutting edges and then running an algorithm for small hop distances.) The main idea was to apply results from the rich field of spanners and emulators, which shows that one can approximate all distances in a graph with a small number of edges. We modified this result to show that a small number of shortcut edges can approximate all distances while effectively maintaining short hop-lengths as well: that is, for any pair (x, y), one can always patch together an approximate shortest path from x to y using just a small number of these shortcut edges. We still had to maintain the distances of the shortcut edges directly, but these shortcut distances were only a small subset of all distances.

We were previously unable to apply the idea of shortcuts to *directed* graphs, however, because in this case it is essentially impossible to approximate all-pairs shortest distances with a sparse spanner or emulator. The key feature of undirected graphs is that if u and v are nearby, then shortest paths from u are approximately the same as those from v, and we can therefore handle a whole cluster of nearby vertices with a single representative. Such clustering does not work in directed graphs because a short $u - v$ path does not imply a short $v - u$ path.

Shortcuts are thus much more difficult to apply in directed graphs, and as far as we know, this is the first paper to do so in the dynamic setting. (In the static setting, Thorup's algorithm for distance oracles in planar graphs [18] uses them extensively, and there are several papers that use them to achieve faster running times in practice; see [1] for an overview.) Because *directed* graphs do not allow for clustering, we end up having to maintain shortcut edges for essentially all pairs, which seems to bring us back to the original predicament of trying to maintain APSP by first maintaining APSP. The key lies in doing the computation in the proper order. The h-SSSP algorithm already provides an efficient way to maintain shortest paths of small hop-length, so we start by shortcutting those. This reduces the hop-length of the other paths because shortcutting a subpath of some path $\pi(x, y)$ reduces $h(x, y)$. The reduced hop-lengths allow h-SSSP to efficiently maintain a larger set of distances, which in turn leads to more shortcut edges and a further reduction in hop-lengths. In iterating this process, we continually shortcut the small-hop subpaths of large-hop paths, to the point where the latter themselves become small-hop and easy to shortcut.

This paper is substantially less technical than our papers which applied the shortcut edge approach to undirected graphs, because we do not rely on the heavy machinery of emulators and clustering. One advantage of this is that it forces us to explore the limits of the core approach itself; directed graphs do not seem to offer much structure, so all we can really do is iteratively use the basic h-SSSP algorithm to create more and more shortcut edges. We now present some of the key lemmas and definitions used throughout our algorithm.

Definition 4.1. *We say that an edge (u, v) with weight $w(u, v)$ is an exact shortcut edge if $w(u, v) = \delta(u, v)$. We say that it is an α-shortcut if $\delta(u, v) \leq w(u, v) \leq (1+\epsilon)^\alpha \delta(u, v)$.*

Definition 4.2. *Let G^* be the graph G with some shortcut edges added. Given some $x-y$ path P, we define the G^*-α-reduction of P to be the $x-y$ path of smallest hop-length whose edges are either part of P itself, or α-shortcuts of subpaths of P. We refer to the hop-length of this reduction as the G^*-α-reduced hop-length of P.*

It is easy to see that the G^*-α-reduction of a shortest path $\pi(x, y)$ is a $(1+\epsilon)^\alpha$-approximate shortest path. If all the shortcut edges were exact, then the reduced path would have the same length as the original one. As is, all the shortcut edges are off by a factor of at most $(1+\epsilon)^\alpha$, so the overall weight is off by at most $(1+\epsilon)^\alpha$.

Lemma 4.3. *If the G^*-α-reduction of some path $\pi(x, y)$ has fewer than h edges, then running the h-SSSP algorithm on G^* up to hop-length h yields a $(1+\epsilon)^{\alpha+1}$-approximation to $\delta(x, y)$.*

Proof. Since the G^*-α-reduction of $\pi(x, y)$ has fewer than h edges, we know that $\delta^h_{G^*}(x, y) \leq \delta(x, y)(1+\epsilon)^\alpha$. But by Theorem 3, the h-SSSP algorithm yields a $1+\epsilon$ approximation to $\delta^h_{G^*}(x, y)$, which is a $(1+\epsilon)^{\alpha+1}$ approximation to $\delta(x, y)$. ☐

We now present a well-known sampling lemma that is used throughout our algorithm.

Lemma 4.4. *Let S be a set of r vertices chosen uniformly at random from V, and let P be some path in G with at least $cn\ln(n)/r$ vertices (c is a constant of our choosing). Then, with probability at least $1 - n^{-c}$, the path P contains at least one vertex in S.*

Proof. For any particular vertex $v \in P$, we have that $Pr[v \in S] = r/n$. Thus,

$$Pr[S \bigcap P = \emptyset] \leq (1 - r/n)^{|P|} \leq (1 - r/n)^{cn\ln(n)/r} < n^{-c}. \qquad ☐$$

For simplicity of presentation, we will fix $c = 9$. The following corollary is then a direct consequence of the union bound.

Corollary 4.5. *Let S be a set of r vertices chosen uniformly at random from V, and let $\mathcal{P}$ be a set of $\leq n^4$ paths in G, each of which contains at least $9n\ln(n)/r$ vertices. Then, with probability at least $1 - n^{-5}$, every path in $\mathcal{P}$ contains at least one vertex in S.*

Remark. Our algorithm only requires the above sampling lemma to hold for $O(n^3 \log(n))$ paths (n^2 shortest paths in $n\log(n)$ different graphs used by our algorithm), so we can use the above corollary. Now, note that since we are assuming an adversary that is oblivious to our random choices, the set of r sampled vertices will be random from the perspective of each version of the graph throughout the update sequence. Thus, Corollary 4.5 holds with probability at least $1 - n^{-5}$ for each version of the graph, so by the union bound, it will hold with high probability for *all* versions within the first $O(n^4)$ updates.

Now, as we discuss in the next few sections, since we are looking for a $(1+\epsilon)$ approximation we only need to register a change to an edge weight when it has increased by at least a $(1+\epsilon)$ factor. The total number of updates that our algorithm actually registers (instead of simply throwing away) is thus $O(m\log(nR))$. On the reasonable assumption that $\log R \leq n^3/m$, the number of updates is $O(n^4)$, so by the above discussion setting $c = 9$ will ensure that the corollary holds with high probability throughout all versions of th graph. More generally, it is not hard to see that if $\log R = O(n^x)$, then setting $c = x+5$ is sufficient. In the extremely unlikely case that $\log R$ is not polynomial in n we would need to set c to be $O(\log\log(R)/\log(n))$, and the running time of the whole algorithm would be multiplied by this factor. In this case, however, our running time is already superpolynomial, and we would likely be better off using an algorithm that works for general weights. All in all *we assume for the rest of the paper that Corollary* 4.5 *holds throughout all versions of the graph.*

5. An $\widetilde{O}(mn^{4/3}\log R/\epsilon)$ algorithm. We now present an algorithm for decrementally maintaining $(1+\epsilon)$ shortest paths in directed graphs with a total update time of $\widetilde{O}(mn^{4/3}\log R/\epsilon)$. We later improve this to $\widetilde{O}(mn\log R/\epsilon)$, but even this preliminary approach already yields the first efficient decremental algorithm for polynomial weights, and for sparse m it even beats the previous state of the art of $\widetilde{O}(n^2\sqrt{m}/\epsilon)$ for *unweighted* directed graphs.

We maintain approximate APSP by separately maintaining distances from different sources using the h-SSSP algorithm. The h-SSSP algorithms that we use are grouped into three distinct layers. The first layer of h-SSSP algorithms runs on the main graph G, and maintains approximate distances from only a small number of sources, up to a limited h. We use these distances to construct shortcut edges, which reduce hop-lengths in G. Our second layer runs on this new graph with shortcut edges added and is thus able to efficiently maintain a larger subset of approximate shortest distances. We use these distances to create even more shortcut edges, which further reduce hop-lengths. Our third and final layer computes all-pairs shortest distances by running h-SSSP from every vertex v; this remains efficient because, thanks to the shortcut edges from the second layer, we only have to run h-SSSP up to a small hop-length h.

Recall that the h-SSSP algorithm is not a one-time computation but rather maintains distances dynamically over *all* updates, so all our overall algorithm needs to do is set up the necessary h-SSSP algorithms in the very beginning and let them run. Each of the three layers is responsible for maintaining its own distance matrix, which is simply an aggregate of the distances maintained by all of the h-SSSP algorithms in that layer. As we process our updates, the distances in these matrices will increase, which will lead to weight-increases in the corresponding shortcut edges.

Dependence on Δ. Our primary concern with respect to running time is to maintain distances in total time $\widetilde{O}(mn\log(R))$; the whole apparatus of shortcut edges was developed for this purpose. But a secondary concern is ensuring that every update incurs an additional overhead of only $O(1)$, i.e., that the dependence on Δ is only $O(\Delta)$. We show in section 7.1 that the h-SSSP building block incurs this optimal overhead of $O(1)$ time per update. The problem is that our APSP algorithm runs h-SSSP algorithms from $O(n)$ different sources, which seems to lead to an overhead of $O(n)$ per update. We resolve this problem by noting that although the total number of updates can be arbitrarily large, most of them will only increase weights by a small amount. Any such insignificant update can simply be ignored in $O(1)$ time, i.e., not processed by any of the h-SSSP algorithms. To this end, we define a function which allows us to only register updates that increase the weight by a $(1+\epsilon)$ factor.

DEFINITION 5.1. *For any number x, let Round$_{(1+\epsilon)}(x)$ be $(1+\epsilon)^{\lceil \log_{(1+\epsilon)}(x)\rceil}$ (the smallest power of $(1+\epsilon)$ that is $\geq x$).*

5.1. The algorithm.

Main setup.

1. Use Dijkstra to compute shortest paths from all $v \in V$ in the original graph, before any updates occur.
2. Sample $n^{2/3}$ vertices uniformly at random, and let A be the resulting set.
3. For all $a \in A$, run the h-SSSP algorithm from a up to hop-length $10n^{2/3}$. Recall that this maintains distances from a over all deletions to come. Store the distances maintained in a matrix $D_{A\times A}$, which is initialized with the distances from Step 1. ($D_{A\times A}$ stores approximate distances between nearby vertices in A.)
4. Let G^* be the graph G plus a shortcut edge added for each pair $(a, b) \in A \times A$. Set shortcut (a, b) to have weight $\mathit{Round}_{(1+\epsilon)}(D_{A\times A}[a, b])$.
5. For each $a \in A$, run the h-SSSP algorithm *to a* in G^* up to hop-length $10n^{1/3}\ln(n)$. Store the results in a matrix $D_{V\times A}$, which is initialized with the distances from Step 1. ($D_{V\times A}$ stores approximate distances to all vertices in A.)
6. For each $v \in V$, let G_v be the graph G with a shortcut edge added from v to every vertex in $a \in A$. Set shortcut (v, a) to have weight $\mathit{Round}_{(1+\epsilon)}(D_{V\times A}[v, a])$.
7. For each $v \in V$, run the h-SSSP algorithm from v up to hop-length $10n^{1/3}\ln(n)$ in G_v. Store the results in a matrix $D_{V\times V}$, which is initialized with the distances from Step 1. This is our final distance matrix.

Query(v, w). To approximate $\delta(v, w)$, simply return $D_{V\times V}[v, w]$.

Update step. Our whole algorithm is essentially contained in the h-SSSP algorithms of the main setup. The only catch is that many of these algorithms are not running on the main graph G but on a graph that also contains some shortcut edges. It is crucial for correctness that we dynamically maintain correct distances for these shortcuts (as $\delta(x, y)$ changes, the weight of an $x-y$ shortcut should also change). Here is the order in which we process an update increase-weight$(x, y) : w_{\text{old}}(x, y) \rightarrow w_{\text{new}}(x, y)$ (an edge deletion can be modeled as increasing the weight to ∞).

- If $\mathit{Round}_{(1+\epsilon)}(w_{\text{new}}(x, y)) = \mathit{Round}_{(1+\epsilon)}(w_{\text{old}}(x, y))$, the algorithm simply throws away the update in $O(1)$ time and does not move on to the steps below. Note that because of this, all edge weights in G are effectively only $(1 + \epsilon)$-approximate.
- Else, if $\mathit{Round}_{(1+\epsilon)}(w_{\text{new}}(x, y)) > \mathit{Round}_{(1+\epsilon)}(w_{\text{old}}(x, y))$, input the update increase-weight(x, y) into all of the h-SSSP algorithms from Step 3, which might cause some of the distances maintained in $D_{A\times A}$ to change.
- For all entries for which $\mathit{Round}_{(1+\epsilon)}(D_{A\times A}[a, b])$ has increased, we increase the weight of corresponding shortcut edge (a, b) in G^* (Step 4) to the new $\mathit{Round}_{(1+\epsilon)}(D_{A\times A}[a, b])$.
- Input the original increase-weight(x, y), as well as all the shortcut-edge weight increases in G^* from the previous step, into the h-SSSP algorithms of Step 5. This might cause changes in $D_{V\times A}$.
- For all $v \in V$, for all entries for which $\mathit{Round}_{(1+\epsilon)}(D_{V\times A}[v, a])$ has increased, we increase the weight of corresponding shortcut edge (v, a) in G_v (Step 6) to the new $\mathit{Round}_{(1+\epsilon)}(D_{V\times A}[v, a])$.
- Input increase-weight(x, y), as well as all the shortcut-edge weight increases from the previous step, into the h-SSSP algorithms of Step 7. This might cause some changes to $D_{V\times V}$, which makes sense, since our final distance matrix should be changing over time.

5.2. Running time analysis. The key observation is that the running time for a single update step is just the time to update the distance matrices $D_{A\times A}, D_{V\times A}$, and $D_{V\times V}$ via the h-SSSP algorithms of the main setup. We also have to increase shortcut edge weights, but every such increase corresponds to a change in $D_{A\times A}, D_{V\times A}$, or $D_{V\times V}$ and so can be charged to the h-SSSP algorithms. Thus, the total update time of our algorithm is simply the sum of the total update times of all of its constituent h-SSSP algorithms. We now proceed to analyze this sum.

Note that the rate at which edge-weights are increased may vary greatly depending on whether we are dealing with G, G^*, or G_v; a single update only increases one edge weight in the main graph G, but this can lead to a large number of shortcut-edge weight increases in G_v. All that matters, however, is that since G only sees weight *increases* (by definition of this being a decremental algorithm), G^* and G_v will also only see weight increases; shortcut edge weights are based on distances in G, so since the latter are only getting larger, the same is true of the former. Thus, the algorithm is decremental from the perspective of each graph involved, which allows us to sidestep the analysis of how many updates occur in each graph; all that matters is that in each graph, the h-SSSP algorithm will always have *total* update time $\widetilde{O}(m'h\log R/\epsilon)$, where m' is the number of edges on the graph in question (m' is larger than m when we add shortcut edges).

Technical note. The running time of h-SSSP depends on $\log(nR)$, but the h-SSSP algorithms run on graphs with weighted shortcut edges, and R might be slightly larger in these shortcutted graphs than in the original graph. But since every shortcut edge corresponds to a $(1+\epsilon)$ approximate distance in the original graph, no shortcut will have weight larger than $R' = (1+\epsilon)nR$, so the running time will not be affected because $\log(nR') = O(\log(nR))$. Thus, we just assume the same R throughout.

- Step 3 runs the h-SSSP algorithm from $n^{2/3}$ vertices up to $h = O(n^{2/3})$, which by Theorem 3 yields a total update time of $\widetilde{O}(n^{2/3}mn^{2/3}\log R/\epsilon) = \widetilde{O}(mn^{4/3}\log R/\epsilon)$.
- Step 5 runs the h-SSSP algorithm to $n^{2/3}$ vertices up to $h = \widetilde{O}(n^{1/3})$. Note, however, that the graph G^* has $(m+n^{4/3})$ edges because of the shortcut edges for $A\times A$. This yields a total update time of $\widetilde{O}(n^{2/3}(m+n^{4/3})n^{1/3}\log R/\epsilon) = \widetilde{O}(mn + n^{7/3}) = \widetilde{O}(mn^{4/3}\log R/\epsilon)$.
- Step 7 runs the h-SSSP algorithm from n vertices up to $h = \widetilde{O}(n^{1/3})$. Each graph G_v has $m + n^{2/3} = O(m)$ edges. Thus, the total update time is $\widetilde{O}(nmn^{1/3}\log R/\epsilon) = \widetilde{O}(mn^{4/3}\log R/\epsilon)$.

Dependence on Δ. Our algorithm is built on many h-SSSP algorithms that process many difference edge weight increases: increases to weights of edges in G but also to the various shortcut edges. Let us define any such weight increase $w_{\text{old}}(x,y) \to w_{\text{new}}(x,y)$ to be *significant* if $Round_{(1+\epsilon)}(w_{\text{new}}(x,y)) > Round_{(1+\epsilon)}(w_{\text{old}}(x,y))$. It is clear from the description of our update step that any weight increase which is *not* significant is thrown away in $O(1)$ time and not processed by any h-SSSP algorithms; this is where the $O(\Delta)$ term comes from. We now show that that the total number of times an h-SSSP algorithm processes a *significant* update is $O(mn\log(nR)/\epsilon)$, which is within our desired update bounds.

Recall the definitions of c, C, and R from section 2. The weight of any edge in G ranges from c to C. Every shortcut edge weight is a $(1+\epsilon)$ approximation to some shortest distance in G, so it ranges from c to at most $(1+\epsilon)nC \le 2nC$. Thus, since edge weights only increase, the number of significant updates on any given edge (shortcut edges included) is at most $O(\log_{1+\epsilon}(2nC/c)) = O(\log(nR)/\epsilon)$.

In particular, since the main algorithm runs $O(n)$ h-SSSP algorithms, and every edge in G (i.e., every nonshortcut edge) registers $O(\log(nR)/\epsilon)$ significant updates, the total number of times that an h-SSSP algorithm processes a significant update on an edge in G is $O(mn\log(nR)/\epsilon)$. G^* contains $O(n^{4/3})$ shortcut edges betweeen vertices in A, and any significant update to one of these is processed by the $O(n^{2/3})$ h-SSSP algorithms running on G^*, leading to an additional $O(n^2\log(nR)/\epsilon)$ significant updates processed by an h-SSSP algorithm. Finally, for each vertex v the graph G_v contains $n^{2/3}$ shortcut edges but has only a single h-SSSP algorithm running on it, so the total number of times an h-SSSP algorithm processed a significant update to a shortcut edge in some graph G_v is only $O(n^{5/3}\log(nR)/\epsilon)$. Thus the total number of times that an h-SSSP algorithm processes a significant update is only $O(mn\log(nR)/\epsilon)$; all other updates are insignificant and thrown away in $O(1)$ time.

5.3. Approximation error analysis. Before proceeding, we observe a basic mathematical fact.

LEMMA 5.2. *For any positive real number $\epsilon < 1$ and any nonnegative integer a, we have $(1+\frac{\epsilon}{2a})^a < 1+\epsilon$.*

Proof. Since $\frac{\epsilon}{2a} < 1$, we have $(1+\frac{\epsilon}{2a})^a < (e^{\frac{\epsilon}{2a}})^a = e^{\epsilon/2} < 1+\epsilon$. $\square$

We now prove that the distances stored in $D_{V\times V}$ are $(1+\epsilon)^6$-approximate. Using $\epsilon' = \epsilon/12$, by Lemma 5.2 we get a $(1+\epsilon/12)^6 \leq (1+\epsilon)$-approximation. Recall that $h(a,b)$, $\delta(a,b)$, and so on always refer to the *current* version of G. Recall also that G^* is the graph from Step 4 of the initialization phase, and G_v are the graphs from Step 6.

LEMMA 5.3. *For any pair $(a,b) \in A\times A$, if $h(a,b) \leq 10n^{2/3}$, then there is a 3-shortcut from a to b in G^* (see Definition 4.1 for 3-shortcut).*

Proof. Recall that the weight of the shortcut edge is $Round_{(1+\epsilon)}(D_{A\times A}[a,b])$. By Theorem 3 the h-SSSP algorithm in Step 3 yields a $(1+\epsilon)$-approximation to $\delta^{10n^{2/3}}(a,b) = \delta(a,b)$; however, since edge weights in G are only a $(1+\epsilon)$-approximation to their actual weight (because we only register updates that increase $Round_{(1+\epsilon)}$ $(w(x,y))$), the total approximation factor is $(1+\epsilon)^2$. Thus, $D_{A\times A}[a,b]$ is $(1+\epsilon)^2$-approximate. The $Round_{(1+\epsilon)}$ function on the shortcut weights adds another $(1+\epsilon)$-approximation, leading to $(1+\epsilon)^3$ in total. $\square$

LEMMA 5.4. *For any pair of vertices $(v,a) \in V\times A$, the G^*-3-reduced hop-length of $\pi(v,a)$ is $\leq 10n^{1/3}\ln(n)$ (see Definition 4.2).*

Proof. We prove this by exhibiting a path of hop-length $\leq 10n^{1/3}\log(n)$ that only uses edges of $\pi(v,a)$ and 3-shortcuts of its subpaths. Let b be the first vertex in A on $\pi(v,a)$. By Lemma 4.4, there are at most $9n^{1/3}\ln(n)$ edges between v and b, so we just follow these directly. We now need to find a path from b to a.

We prove the following by induction: for any vertex $a' \in A$, there is a path from a' to a consisting of at most $\lceil h(a',a)/n^{2/3}\rceil$ 3-shortcuts of subpaths of $\pi(a',a)$.

- *Base case.* If $h(a,a') \leq 10n^{2/3}$, then by Lemma 5.3 there is a 3-shortcut from a' to a.
- *Induction step.* We now assume that the claim is true for all vertices $a' \in A$ for which $h(a',a) \leq i$, and prove that it then also holds when $h(a',a) = i+1$. If $h(a',a) \leq 10n^{2/3}$, we use the base case; otherwise, by Lemma 4.4, there is some vertex $a'' \in A$ on $\pi(a',a)$ that is between $n^{2/3}$ and $10n^{2/3}$ vertices away from a' (because this interval contains $9n^{2/3} \geq 9n^{1/3}\ln(n)$ vertices). Since $h(a',a'') \leq 10n^{2/3}$, there exists a 3-shortcut (a',a''); combining this 3-shortcut with the path of at most $\lceil h(a'',a)/n^{2/3}\rceil \leq (\lceil h(a',a)/n^{2/3}\rceil - 1)$

3-shortcuts from a'' to a guaranteed by the induction hypothesis yields the desired path of 3-shortcuts from a' to a.

We can now prove the main lemma: to get from v to a we first use $9n^{1/3}\ln(n)$ nonshortcut edges to get from v to the first vertex b on $\pi(v, A)$ that is in A; since $h(b, a)$ is trivially $\leq n$, we now take a path of at most $\lceil n/n^{2/3}\rceil = \lceil n^{1/3}\rceil$ 3-shortcuts from b to a. ∎

COROLLARY 5.5. *All the entries in $D_{V\times A}$ are $(1+\epsilon)^4$ approximate distances. This follows directly from Lemmas 5.4 and 4.3.*

LEMMA 5.6. *All the entries in $D_{V\times V}$ are $(1+\epsilon)^6$-approximate distances.*

Proof. We maintain $D_{V\times V}$ by running the h-SSSP algorithm on each G_v. We know that all the shortcuts in G_v are 5-shortcuts because their weights are obtained from $D_{V\times A}$; we get a $(1+\epsilon)^4$ error from $D_{V\times A}$ and another $(1+\epsilon)$ from the $Round_{(1+\epsilon)}$ function.

We can now show that the G_v-5-reduced hop-length of any $\pi(v, w)$ is $\leq 10n^{1/3}\log(n)$. If $h(v, w) \leq 10n^{1/3}\log(n)$, then this is trivially true. Otherwise, by Lemma 4.4 there must be a vertex in A on $\pi(v, w)$. Let a be the *last* such vertex, and note that again by Lemma 4.4, $h(a, w) \leq 9n^{1/3}\log(n)$. Thus, our 5-reduced path is just the 5-shortcut from v to a (created in Step 6), followed by the path $\pi(a, w)$. By Lemma 4.3, running h-SSSP up to $h = 10n^{1/3}\log(n)$ yields a $(1+\epsilon)(1+\epsilon)^5 = (1+\epsilon)^6$-approximation. ∎

6. The $\widetilde{O}(mn\log R/\epsilon)$ algorithm.

Our $\widetilde{O}(mn\log R/\epsilon)$ algorithm is a direct extension of the $\widetilde{O}(mn^{4/3}\log R/\epsilon)$ algorithm above. The basic idea remains the same; we maintain some subset of the shortest distances and use these to construct shortcut edges which lower hop-lengths and hence allow us to efficiently maintain more shortest distances, and so on. Once again the h-SSSP algorithms dynamically maintain distances from a source over *all* updates, so we just set them up in the beginning and let them run. In this version, however, instead of using three layers of h-SSSP algorithms, we use $\log(n)$ layers. We assume for simplicity that n is a power of 4.

DEFINITION 6.1. *For the rest of this paper, we define q to be $\log(n)/2$.*

DEFINITION 6.2. *Define A_0 to be V. Construct A_1 by picking half of the vertices from $A_0 = V$ uniformly at random. Construct A_2 by picking half the vertices from A_1 uniformly at random. Keep doing this until we reach $A_q = A_{(\log(n)/2)}$. Note that A_k contains $n/2^k$ random vertices from V and that $|A_q| = \sqrt{n}$.*

6.1. The algorithm.

Main setup.

1. Use Dijkstra to compute, for all vertices v, shortest paths from v in the original graph G before any updates.
2. Construct the sets $A_0, A_1, \ldots, A_q$ as in Definition 6.2.
3. For each $v \in A_q$, run the h-SSSP algorithm to and from v up to $h = 10\sqrt{n}\log(n)$ on the main graph G. Store the results in $D_{A_q\times A_q}$. We initialize this matrix with the distances from Step 1. ($D_{A_q\times A_q}$ contains approximate distances between nearby vertices in A_q.)
4. Let G_q be the graph G with a shortcut edge (v, w) added for each pair $(v, w) \in A_q \times A_q$. Set the weight of shortcut (v, w) to be $Round_{(1+\epsilon)}(D_{A_q\times A_q}[v, w])$.
5. For each $v \in A_q$, run the h-SSSP algorithm *to and from* v on the graph G_q up to $h = 10\sqrt{n}\log(n)$. Store the results in matrix D_q. As before, initialize D_q with the distances from Step 1. (D_q contains approximate distances to and from vertices in A_q.)

6. For $k = q - 1$ down to 0:
   - For each $v \in A_k$, we create a graph $G_{v,k}$, which is the graph G with shortcut edges (v, w) and (w, v) added for every $w \in A_{k+1}$. Set the weight of shortcut (v, w) to be $Round_{(1+\epsilon)}(D_{k+1}[v, w])$, and do the same for (w, v).
   - For each $v \in A_k$, run the h-SSSP algorithm *to and from* v in $G_{v,k}$ up to $h = 10\log(n)2^{k+1}$. Store the combined results for all $v \in A_k$ in matrix D_k. (D_k contains approximate distances to and from vertices in A_k.)

Query(v, w). To find an approximation to any $\delta(v, w)$, simply look up $D_0[v, w]$.

Update step. As in section 5, all the work of our algorithm is done by the various h-SSSP algorithms of the main setup. All we need to describe here is the order in which we process some update increase-weight(x, y) : $w_{\text{old}}(x, y) \to w_{\text{new}}(x, y)$ (an edge deletion can be modeled as increasing the weight to ∞).

- If $Round_{(1+\epsilon)}(w_{\text{new}}(x, y)) = Round_{(1+\epsilon)}(w_{\text{old}}(x, y))$, the algorithm deems the update insignificant and throws it away in $O(1)$ time; i.e., it skips the steps below and does not process the update in any h-SSSP algorithm.
- Else, if $Round_{(1+\epsilon)}(w_{\text{new}}(x, y)) > Round_{(1+\epsilon)}(w_{\text{old}}(x, y))$, input the update increase-weight(x, y) into all of the h-SSSP algorithms from Step 3. This might cause some of the distances maintained in $D_{A_q \times A_q}$ to change.
- For all $(v, w) \in A_q \times A_q$ for which we have that $Round_{(1+\epsilon)}(D_{A_q \times A_q}[v, w])$ has increased, we increase the weight of shortcut edge (v, w) in G_q (Step 4) to the new $Round_{(1+\epsilon)}(D_{A_q \times A_q}[v, w])$.
- Input the original increase-weight(x, y) as well as all the shortcut-edge weight increases in G_q from the previous step into the h-SSSP algorithms of Step 5. This might cause changes to D_q.
- For $k = q - 1$ down to 0:
  - For all pairs $(v, w) \in A_k \times A_{k+1}$ for which $Round_{(1+\epsilon)}(D_{k+1}[v, w])$ or $Round_{(1+\epsilon)}(D_{k+1}[w, v])$ has increased, we increase the weight of corresponding shortcut edge (v, w) or (w, v) in $G_{v,k}$ to the new value of $Round_{(1+\epsilon)}(D_{k+1}[v, w])$ or $Round_{(1+\epsilon)}(D_{k+1}[w, v])$.
  - For each $v \in A_k$, input the original increase-weight(x, y) as well all the shortcut-edge weight increases from the previous step into the h-SSSP algorithm to and from v in $G_{v,k}$. Record the changed distances for each v into the matrix D_k.

6.2. Running time analysis. As in section 5.2, we have various graphs $G_{v,k}$ whose edges are changing at different rates, but the algorithm is nonetheless decremental from the point of view of each of these graphs. Thus, we simply need to analyze the total update times of all the h-SSSP algorithms and the time to maintain G_q and all the $G_{v,k}$. Recall the definitions of c, C, and R from section 2. (Technical note: The value of R might vary slightly among the h-SSSP algorithms because of the shortcut weights, but, as discussed in section 5.2, it never gets so big as to asymptotically affect the running time.)

- To maintain $D_{A_q \times A_q}$ in Step 3, we run the h-SSSP algorithm from $\sqrt{n}$ vertices up to $h = O(\sqrt{n}\log(n))$, which results in total update time $O((\sqrt{n}\log(n))(m\sqrt{n}\log(n)\log(nR)/\epsilon)) = O(mn\log^2(n)\log(nR)/\epsilon)$.
- For all k, we maintain D_k by running the h-SSSP algorithm from $|A_k| = n/2^k$ vertices up to $h = O(2^k \log(n))$. Each graph $G_{v,k}$ has $O(m + |A_{k+1}|) = O(m)$ edges, so the total update time corresponding to any given k is $O((n/2^k) \cdot (2^k \log(n)) \cdot (m\log(n)\log(nR)/\epsilon)) = O(mn\log^2(n)\log(nR)/\epsilon)$. There are

$O(\log(n))$ values of k, which yields another log factor. Finally, as we discuss at the beginning of the next section, in order for the algorithm to yield the desired $(1+\epsilon)$ approximation it must internally use $\epsilon' = \epsilon/\log(n)$, which incurs an additional $\log(n)$ factor. The final running time is thus $O(mn\log^4(n)\log(nR)/\epsilon)$. The same analysis yields a slightly faster running time of $O(mn\log^4(n)\log\log(n)/\epsilon)$ in unweighted graphs because the h-SSSP algorithm is slightly faster in that case (see Theorem 3).

Dependence on Δ. The analysis of the overhead per update is almost identical to that of section 5.2. As before, the number of significant updates on any given edge (shortcut edges included) is at most $O(\log_{1+\epsilon}(2nC/c)) = O(\log(nR)/\epsilon)$. The total number of h-SSSP algorithms is $|A_q| = \sqrt{n}$ running on the original graph G, $|A_q| = \sqrt{n}$ running on G_q, and one more for each graph $G_{v,k}$, for a total of $2\sqrt{n} + \sum_{k=0}^{q}(n/2^k) = O(n)$. Thus, the total number of times that an h-SSSP algorithm processed a significant update on an edge in G (i.e., a nonshortcut edge) is $O(mn\log(nR)/\epsilon)$.

We now bound the number of times an h-SSSP algorithm processed a significant update on a shortcut edge. The graph G_q has $O(n)$ shortcut edges, and there are $O(\sqrt{n})$ h-SSSP algorithms running on it, for a total of $O(n^{1.5}\log(nR)/\epsilon)$ significant updates processed. Each graph $G_{v,k}$ has $|A_{k+1}| = n/2^{k+1} = O(n)$ shortcut edges (see section 6.2 for details), each of which is processed by one single h-SSSP algorithm: the one running to and from v in $G_{v,k}$. For any fixed k, there are $|A_k| = O(n/2^k)$ graphs $G_{v,k}$, so the total number of shortcut edges over all the $G_{v,k}$ graphs is $O(n\sum_{k=1}^{q} n/2^k) = O(n^2)$. Since each edge registers only $O(\log(nR)/\epsilon)$ significant updates, the total number of significant shortcut weight increases processed by our algorithm, i.e., the number of times that a h-SSSP algorithm must handle a shortcut weight increase, is only $O(n^2\log(nR)/\epsilon)$. All in all, combining shortcut and nonshortcut edges, the total number of times that an h-SSSP algorithm processes a significant update is only $O(mn\log(nR)/\epsilon)$; all other updates are insignificant and thrown away in $O(1)$ time.

6.3. Approximation analysis. We will prove that our final distance matrix $D_0[v,w]$ contains a $(1+\epsilon)^{(4+\log(n))}$-approximation to all shortest distances. Using $\epsilon' = \epsilon/(4\log(n))$, we get a $(1+\frac{\epsilon}{4\log(n)})^{(4+\log(n))} \le (1+\frac{\epsilon}{4\log(n)})^{2\log(n)} \le (1+\epsilon)$-approximation (see Lemma 5.2) while only multiplying the running time by $O(\log(n))$. Generally speaking, each layer of the algorithm incurs a $(1+\epsilon)^2$-approximation: one $(1+\epsilon)$ factor comes from the h-SSSP algorithm, while the other comes from applying the $Round_{(1+\epsilon)}$ function to the shortcut weights of the graphs in that layer.

The graphs G_q and $G_{v,k}$ refer to the graphs created during the main setup (see section 6.1). Recall that $\delta(x,y), h(x,y)$, and $\pi(x,y)$ are changing over time, and refer to the *current* graph.

LEMMA 6.3. *For any pair* $(x,y) \in V \times A_q$, *the entries* $D_q[x,y]$ *and* $D_q[y,x]$ *are* $(1+\epsilon)^4$*-approximations to* $\delta(x,y), \delta(y,x)$, *respectively.*

Proof. First note that $D_{A_q\times A_q}$ is maintained by running the h-SSSP algorithm on the main graph G, so it is $(1+\epsilon)$-approximate up to $h = 10\sqrt{n}\log(n)$. However, since edge weights in G are themselves only $(1+\epsilon)$-approximate (because we only register an update to $w(x,y)$ if it increases $Round_{(1+\epsilon)}(w(x,y)))$, h-SSSP returns $(1+\epsilon)^2$-approximate distances. Thus, for any pair $(a,b) \in A_q \times A_q$ with $h(a,b) \le 10\sqrt{n}\log(n)$, there is a 3-shortcut edge (a,b) in G_q; it is a 3-shortcut rather than a 2-shortcut because of the extra $(1+\epsilon)$ error that comes from applying the $Round_{(1+\epsilon)}$ function to the shortcut weight.

We now prove that the G_q-3-reduction of $\pi(x,y)$ has at most $10\sqrt{n}\log(n)$ edges. Since we run each h-SSSP algorithm to and from its source, the proof for $\pi(y,x)$ is exactly the same. More generally, the proof is completely analogous to that of Lemma 5.4. The goal is to exhibit an $x-y$ path P of hop-length $\leq 10\sqrt{n}\log(n)$ that uses only edges of $\pi(x,y)$ and 3-shortcuts of its subpaths. Recall that $y \in A_q$, and let x_2 be the first vertex in A_q on $\pi(x,y)$. By Lemma 4.4, x_2 is at most $9\sqrt{n}\log(n)$ vertices away from x, so our path P will just directly take the subpath $\pi(x,x_2)$.

To get from x_2 to y, we prove the following by induction: given any $y' \in A_q$, there is a path from y' to y consisting of at most $\lceil h(x,y)/\sqrt{n}\rceil$ 3-shortcuts of subpaths of $\pi(y',y)$.

- *Base case.* If $h(y',y) \leq 10\sqrt{n}\log(n)$, then G_q contains a 3-shortcut between them, so we just use that.
- *Induction step.* We now assume that the claim holds for all vertices $y' \in A_q$ for which $h(y',y) \leq i$, and prove it for the case that $h(y',y) = i+1$. If $h(y',y) \leq 10\sqrt{n}\log(n)$, we simply use the base case; otherwise, again by Lemma 4.4, $\pi(y',y)$ contains some vertex $y'' \in A_q$ that is between $\sqrt{n}$ and $10\sqrt{n}\log(n)$ vertices away from y' (this interval of vertices is a shortest path with more than $9\sqrt{n}\log(n)$ vertices, so it must contain a vertex in A_q). Since $h(y',y'') \leq 10\sqrt{n}\log(n)$, G_q contains a 3-shortcut (y',y''); combining this 3-shortcut with the path of $\lceil h(y'',y)/\sqrt{n}\rceil \leq (\lceil h(y',y)/\sqrt{n}\rceil - 1)$ 3-shortcuts from y'' to y guaranteed by the induction hypothesis yields the desired path of 3-shortcuts from y' to y.

Our final path from x to y consists of the at most $9\sqrt{n}\log(n)$ nonshortcut edges from x to x_2, followed by the $\lceil h(x_2,y)/\sqrt{n}\rceil \leq \lceil n/\sqrt{n}\rceil = \lceil\sqrt{n}\rceil$ 3-shortcuts from x_2 to y, yielding fewer than $10\sqrt{n}\log(n)$ edges in total. By Lemma 4.3 this implies that the h-SSSP algorithm of Step 5, which runs up to hop-length $h = 10\sqrt{n}\log(n)$, returns a $(1+\epsilon)(1+\epsilon)^3 = (1+\epsilon)^4$-approximation, as desired. □

LEMMA 6.4. *Given any nonnegative integer $k \leq q = \log(n)/2$ and any pair $(u,v) \in A_k \times V$, we have that $D_k[u,v]$ and $D_k[v,u]$ are $(1+\epsilon)^{4+2(q-k)}$-approximations to $\delta(u,v)$, $\delta(v,u)$. In particular, for any pair of vertices $u,v \in V$, $D_0[u,v]$ is a $(1+\epsilon)^{\log(n)+4}$-approximation, as desired.*

Proof (by induction). We proved the base case of $k=q$ in Lemma 6.3, so only the induction step is left. We assume the lemma is true for some k and prove that it also holds for $k-1$.

For any $v \in A_{k-1}$, all the shortcut edges in $G_{v,k-1}$ come from D_k, so since the lemma holds for D_k, these must all be $(5+2(q-k))$-shortcuts; the extra $(1+\epsilon)$ factor (from 4 to 5) comes from the application of the $Round_{(1+\epsilon)}$ function to the shortcut weights in $G_{v,k-1}$. We now show that for any pair $(u,v) \in V \times A_{k-1}$, the $G_{v,k-1}$-$(5+2(q-k))$-reduction of $\pi(u,v)$ has hop-length $\leq 10\log(n)2^k$. Let u_2 be the first vertex in A_k on $\pi(u,v)$ (if u_2 does not exist, then by Lemma 4.4 $h(u,v) \leq 9\log(n)2^k$, so we are done). We know from Lemma 4.4 that the subpath $\pi(u,u_2)$ of $\pi(u,v)$ contains at most $9\log(n)2^k$ edges. Moreover, because of how we constructed $G_{v,k-1}$, there must be a $(5+2(q-k))$-shortcut (u_2,v). We have thus exhibited a $u-v$ path with $\leq 9\log(n)2^k + 1 \leq 10\log(n)2^k$ edges, as desired. It is not hard to see that by symmetry, the same holds for the reverse direction: for any $(u,v) \in V \times A_{k-1}$ the $G_{v,k-1}$-$(5+2(q-k))$-reduction of $\pi(v,u)$ has hop-length $\leq 10\log(n)2^k$.

Thus, by Corollary 4.3, the h-SSSP algorithm to and from v on $G_{v,k-1}$ up to $h = 10\log(n)2^k$ incurs an additional $(1+\epsilon)$-aproximation and returns a $(1+\epsilon)^{6+2(q-k)} = (1+\epsilon)^{4+2(q-(k-1))}$-approximation to both $\delta(u,v)$ and $\delta(v,u)$. Our argument holds for all pairs $(u,v) \in V \times A_{k-1}$, so we are done. □

7. The h-SSSP algorithm.

We now present the h-SSSP algorithm for decrementally maintaining an approximate shortest path tree up to hop-length h. This algorithm is not new to this paper and was used as a subroutine in Bernstein's FOCS 2009 paper [3]. Recall the main theorem we are trying to prove.

Theorem 4 (see [3]). *Given a source s and a hop distance h, We can decrementally maintain distances $\delta'(s,v)$ to every vertex v such that we always have $\delta(s,v) \leq \delta'(s,v) \leq (1+\epsilon)\delta^h(s,v)$. The total update time over all deletions and weight-increases is $O(mh\log(n)\log(nR)/\epsilon + \Delta)$ in weighted graphs and $O(mh\log(n)\log\log(n))$ in unweighted ones.*

Recall from Theorem 2 that the main idea behind King's $O(md)$ algorithm was to only explore the edges of a vertex v when the distance to v from s changed. The basic idea of our $(1+\epsilon)$-approximation is to only explore the edges of v when $\delta(s,v)$ changes by a significant amount. The h-SSSP algorithm is actually broken up into many smaller algorithms, each of which handles different ranges of $\delta^h(s,v)$.

Definition 7.1. *Given a source vertex s, a hop-length h, and an integer k, we say that algorithm A_k maintains h-SSSP$_k$ if it decrementally maintains distances $\delta'_k(s,v)$ with the following properties:*

- *If $2^k \leq \delta^h(s,v) \leq 2^{k+1}$, then $\delta(s,v) \leq \delta'_k(s,v) \leq (1+\epsilon)\delta^h(s,v)$.*
- *Otherwise, our only guarantee is that $\delta(s,v) \leq \delta'_k(s,v)$.*

Lemma 7.2. *Assuming $0 < \epsilon < 1$, we can maintain any h-SSSP$_k$ in total update time $O(mh/\epsilon + \Delta)$.*

Proof of Lemma 7.2. Recall that here we are only concerned with approximating distances $\delta^h(s,v)$ for which $2^k \leq \delta^h(s,v) \leq 2^{k+1}$. For the rest of this proof, let $\alpha = \frac{\epsilon 2^k}{h}$. We start by scaling the edge-weights of graph G to obtain a new graph G_k in the following way:

- Delete all edges of weight $> 2^{k+1}$ from G.
- Round all remaining edge weights up to the nearest integer multiple of α.
- Divide all edge weights by α.

Note that the scaled weights in G_k are positive and integral. Our algorithm maintains a shortest path tree from s in G_k by simply running King's $O(md)$ decremental SSSP algorithm (see section 3) up to distance $d = \lceil 4h/\epsilon \rceil$. More precisely, if an update deletes (u,v) in the original graph, we simply delete (u,v) in the scaled graph; if the update is increase-weight (u,v), we first scale the new weight according to the three steps above and then change the weight of $w(u,v)$ to this new scaled weight. We then output $\delta'_k(s,v) = \alpha \cdot \delta_{G_k}(s,v)$. By Corollary 3.1, the total update time of King's algorithm is just $O(md) = O(mh/\epsilon)$, as desired. The final $O(\Delta)$ term arises from weight increases in G that do not change the scaled weights in G_k (i.e., the old weight and the new weight scale up to the same nearest multiple of α) and so are discarded in $O(1)$ time. We now do an approximation analysis.

Let G^*_k be the graph G_k before dividing the edge weights by α (but after scaling up to a multiple of α), and note that since G_k and G^*_k are the same up to a scaling factor, our output is precisely $\delta'_k(s,v) = \alpha \cdot \delta_{G_k}(s,v) = \delta_{G^*_k}(s,v)$. All edge weights in G^*_k are greater than those in G, so it is clear that $\delta'_k(s,v) = \delta_{G^*_k}(s,v) \geq \delta(s,v)$. We now need to show that if $2^k \leq \delta^h(s,v) \leq 2^{k+1}$, then $\delta_{G^*_k}(s,v) \leq (1+\epsilon)\delta^h(s,v)$. To see this, let us examine how the weight changes $G \rightarrow G^*_k$ affect $\pi^h(s,v)$. Since $\delta^h(s,v) \leq 2^{k+1}$, $\pi^h(s,v)$ does not contain any edges of weight $> 2^{k+1}$, so the first set of changes does not affect it at all. The second set of changes adds up to α weight to every edge on $\pi^h(s,v)$, so the weight of the path in G^*_k is at most $\delta^h(s,v) + h\alpha = \delta^h(s,v) + \epsilon 2^k \leq (1+\epsilon)\delta^h(s,v)$ (the last inequality follows from $2^k \leq \delta^h(s,v)$). Thus, we have exhibited an $s - v$ path in G^*_k of weight $\leq (1 + \epsilon)\delta^h(s, v)$, so certainly the *shortest* $s - v$ path in G^*_k will have weight $\leq (1 + \epsilon)\delta^h(s, v)$, as desired. Finally, note that the weight of this path in G_k is at most $(1 + \epsilon)\delta^h(s, v)/\alpha \leq \frac{2 \cdot 2^{k+1}}{\epsilon 2^k/h} = 4h/\epsilon$, so running King's algorithm up to $d = \lceil 4h/\epsilon \rceil$ in G_k will in fact find this path. ▯

We now show how to obtain an algorithm for h-SSSP by simply combining h-SSSP$_k$ algorithms for different values of k. The most natural way to do this, however, achieves a slightly worse dependence on $O(\Delta)$ than the one promised in Theorem 4: $O(\Delta \log(nR) \log\log(nR))$. For the sake of intuition, we show in this section a simple method for achieving this worse update time, and then show in the next section how to reduce the dependence on Δ to $O(\Delta)$. Recall the definitions of c, C, and R from section 2.

LEMMA 7.3. *If for any k we could maintain h-SSSP$_k$ in total update time T, then we would have an algorithm for h-SSSP$_k$ with total update time $O(T \log(nR)$ $\log\log(nR))$.*

Proof. All we do is maintain h-SSSP$_k$ for $\lfloor \log(c) \rfloor \leq k \leq \lceil \log(nC) \rceil$. The crux is that if we then set $\delta'(s, v) = \min\{\delta'_k(s, v)\}$, we have the desired property $\delta(s, v) \leq \delta'(s, v) \leq (1 + \epsilon)\delta^h(s, v)$. To see this, note that $\delta(s, v)$ can never be smaller than c or larger than nC, so, in particular, there is some k between $\lfloor \log(c) \rfloor$ and $\lceil \log(nC) \rceil$ for which $2^k \leq \delta^h(s, v) \leq 2^{k+1}$. For this value of k, we know that h-SSSP$_k$ outputs $\delta'_k(s, v) \leq (1 + \epsilon)\delta^h(s, v)$, so it is certainly true that $\delta'(s, v) = \min_k\{\delta'_k(s, v)\} \leq (1 + \epsilon)\delta^h(s, v)$. That $\delta'(s, v) \geq \delta(s, v)$ follows from the fact that every $\delta'_k(s, v)$ is $\geq \delta(s, v)$.

Thus, for each vertex v, our algorithm maintains a min-heap of all $\delta'_k(s, v)$ for $\lfloor \log(c) \rfloor \leq k \leq \lceil \log(nC) \rceil$. A query operation for $\delta'(s, v)$ simply returns the minimum of this heap in $O(1)$ time. An update operation on edge (x, y) is inputted into every h-SSSP$_k$, and whenever some $\delta'_k(s, v)$ changes, we update the min-heap for $\delta'(s, v)$ in $O(\log\log(nR))$ time. We maintain $O(\log(nR))$ different h-SSSP$_k$, so the total time spent processing updates in the h-SSSP$_k$ is $O(T \log(nR))$. This is also the bound on how often some $\delta'_k(s, v)$ can change, so the total time updating the min-heap is $O(T \log(nR) \log\log(nR))$. Together the two add to $O(T \log(nR) \log\log(nR))$, as desired. ▯

COROLLARY 7.4. *We can maintain h-SSSP in total time $O(mh \log(nR) \log\log \cdot (nR)/\epsilon + \Delta \log(nR) \log\log(nR))$. This follows directly from the two preceding lemmas. Note that in unweighed graphs $R = 1$ and $\Delta \leq m$, and so the running time is $O(mh \log(n) \log\log(n)/\epsilon)$, as promised in Theorems 3 and 4.*

7.1. Limiting the dependence on Δ to $O(\Delta)$. In this section, we improve the dependence on Δ in Corollary 7.4 to $O(\Delta)$, thus achieving the total update time for unweighted graphs promised in Theorems 2 and 4. Note that although the h-SSSP algorithm itself was used in Bernstein's FOCS 2009 paper [3], in that paper the implicit dependence on Δ was $O(\Delta \log(nR) \log\log(nR))$. *Thus, the reduction to an $O(\Delta)$ dependence on Δ is in fact new.*

That being said, the improvement in this subsection is extremely technical and not particularly interesting from a conceptual perspective. Moreover, it is not all that important: using the simple h-SSSP algorithm presented in Lemma 7.3 and Corollary 7.4 would yield a decremental APSP algorithm with total update time $\widetilde{O}(mn \log R/\epsilon) + O(\Delta \log(nR) \log\log(nR))$. The $O(\Delta \log(nR) \log\log(nR))$ term is very unlikely to affect the asymptotic running time, especially as if it ever came to dominate that would imply that we were achieving an amortized update time of

$O(\log(nR)\log\log(nR))$, which is already very good. The reader would thus not lose much in simply skipping the current section.

We now continue with the improvement to h-SSSP. Say that we are running the h-SSSP algorithm up to hop-length h. Let us focus on processing a particular update increase-weight(u, v), and let $w_{\text{old}}(u, v)$ and $w_{\text{new}}(u, v)$, respectively, correspond to the weights of (u, v) before and after the update (a deletion can be modeled by setting $w_{\text{new}}(u, v) = \infty$). Since we are in a decremental setting, we know that $w_{\text{old}}(u, v) < w_{\text{new}}(u, v)$. The naive way for h-SSSP to process increase-weight(u, v) is to process this update in each of the h-SSSP$_k$ algorithms: there are $O(\log(nR))$ different values for k, so this would require a minimum of $O(\log(nR))$ time. But note that there is no reason to process this update in some particular h-SSSP$_k$ if we know that increase-weight(u, v) has no chance of affecting $\delta'_k(s, x)$ for any vertex x. Thus, our basic approach is to only update increase-weight(u, v) in those h-SSSP$_k$ for which it might be relevant.

Before proceeding, let us carefully pinpoint the different steps taken by the h-SSSP algorithm, so that we can analyze the parts separately. The algorithm for handling an update increase-weight(u, v) can be thought of as consisting of the three operations below.

Running time breakdown.

1. Figure out for which h-SSSP$_k$ the update might be relevant, and register the update in those h-SSSP$_k$ only.
2. Process the update in the chosen h-SSSP$_k$, thus potentially changing various $\delta'_k(s, v)$. This is where the "real work" occurs.
3. Now that some of the $\delta'_k(s, v)$ have changed, we must update $\delta'(s, v) = \min_k\{\delta'_k(s, v)\}$.

We have already analyzed the total time spent in Step 2 over all updates: each h-SSSP$_k$ spends a total of $O(mh/\epsilon)$ time processing its updates, so since there are $O(\log(nR))$ possible values of k, among all h-SSSP$_k$ we have total update time $O(mh\log(nR)/\epsilon)$, as desired. For Steps 1 and 3 to be efficient, however, we must modify the h-SSSP algorithm.

We start with Step 3. First let us bound how often the $\delta'_k(s, v)$ might change. Recall that h-SSSP$_k$ runs on a scaled graph G_k, where it only stores distances up to distance $\lceil 4h/\epsilon\rceil$. Thus, since distances only increase, it is clear that for any particular v, $\delta'_k(s, v)$ can change at most $\lceil 4h/\epsilon\rceil$ times. Summing over all vertices v, and all the h-SSSP$_k$, we see that in total over all updates there are $O(nh\log(R)/\epsilon)$ changes to the $\delta'_k(s, v)$. The total time spent in Step 3 is thus $O([nh\log(nR)/\epsilon]\cdot$ [the time to update $\delta'(s, v) = \min_k\{\delta'_k(s, v)\}$ when some $\delta'_k(s, v)$ changes]).

The second term of course depends on our data structure for updating $\delta'(s, v) = \min_k\{\delta'_k(s, v)\}$. The most natural option would be, for any particular v, to store all the $\delta'_k(s, v)$ in a min-heap. This heap would have $O(\log(nR))$ elements and so update time $O(\log\log(nR))$. This is not quite good enough, as it would imply a total time of $O(nh\log(nR)\log\log(nR)/\epsilon)$ spent in Step 3, which is not strictly contained in our desired bound of $O(mh\log(n)\log(nR)/\epsilon)$. We now present a different data structure.

LEMMA 7.5. *Given any vertex v, and assuming that $(1/\epsilon)$ is at most polynomial in n, we can build a data structure on the $\delta'_k(s, v)$ that returns $\delta'(s, v) = \min_k\{\delta'_k(s, v)\}$ in $O(1)$ time and processes an increase to some $\delta'_k(s, v)$ in $O(\log(n))$ time.*

Proof. The data structure is based on the following observation.

Observation. Let k' be some index for which $\delta'_{k'}(s, v) \neq \infty$. Then, for any index $k^* > k' + 2 + \log(h/\epsilon)$ we always have that $\delta'(s, v) \neq \delta'_{k^*}(s, v)$.

This observation relies on the details of the h-SSSP$_k$ algorithm presented in the proof of Lemma 7.2. For our index k', h-SSSP$_{k'}$ only runs up to distance $\lceil 4h/\epsilon \rceil$ on the scaled graph $G_{k'}$, so any path it finds will have length at most $\lceil 4h/\epsilon \rceil$ in $G_{k'}$. It is clear from how h-SSSP$_k$ performs the scaling that any edge weight in G'_k is no larger than the corresponding edge weight in G divided by $\epsilon 2^{k'}/h$; thus, the unscaled length in G of any path found by h-SSSP$_k$ is at most $(\lceil 4h/\epsilon \rceil) \cdot (\epsilon 2^{k'}/h) \leq 2^{k'+2} + 2^{k'} < 2^{k'+3}$, so $\delta'_{k'}(s,v) \neq \infty$ implies that $\delta'(s,v) \leq \delta'_{k'}(s,v) < 2^{k'+3}$. But now, looking at k^*, we see that we always have $\delta'_{k^*}(s,v) \geq \epsilon 2^{k^*}/h$ because we scale each edge weight up to the nearest multiple of $\epsilon 2^{k^*}/h$. Thus, if $k^* > k' + 2 + \log(h/\epsilon)$, then since k^* is an integral index we have $k^* \geq k' + 3 + \log(h/\epsilon)$, so $\delta'_{k^*}(s,v) \geq \epsilon 2^{k^*}/h \geq 2^{k'+3} > \delta'(s,v)$, so $\delta'(s,v) \neq \delta'_{k^*}(s,v)$, as desired.

Data structure. The data structure is very simple: we keep the $\delta'_k(s,v)$ in a linked list, sorted in increasing order of k (*not* in increasing order of $\delta'_k(s,v)$). We also maintain a pointer to the minimum $\delta'_k(s,v)$ in the list. We can return $\delta'(s,v)$ in $O(1)$ time by following the min-value pointer, so we now focus on updates to the $\delta'_k(s,v)$. We will always throw away any $\delta'_k(s,v)$ for which $\delta'_k(s,v) = \infty$, so the head of the list will be the first entry for which $\delta'_k(s,v) \neq \infty$. Thus, by the observation above we know that $\delta'(s,v) = \min_k\{\delta'_k(s,v)\}$ will always be within $k' + 2 + \log(h/\epsilon) = O(\log(n))$ entries from the head (we are assuming that $(1/\epsilon)$ is at most polynomial in n). After some increase-weight(x,y), some of the $\delta'_k(s,v)$ values may be increased. To process these increases, we first update the $\delta'_k(s,v)$ in our list (we can trivially maintain pointers that will allow us to do this). We then go through the list, starting from the head, and delete any $\delta'_k(s,v)$ that has come to equal ∞; since distances only increase, once $\delta'_k(s,v) = \infty$ for some index k, it will continue to be ∞ in the future, so we can safely throw it away. We stop when we reach the first $\delta'_k(s,v) \neq \infty$. By the observation above we can now find the minimum by comparing this first entry and the $k' + 2 + \log(h/\epsilon) = O(\log(n))$ entries that come after it, and then update our min-value pointer. It is clear that the update time per $\delta'_k(s,v)$-increase is just $O(\log(n) + [\text{number of } \delta'_k(s,v) \text{ deleted}])$. For a fixed v, the second term amounts to a total of $\mathrm{O}(\log(nR))$ over all updates, since that is the number of possibilities for k. Summed over all vertices v, this yields an extra $O(n\log(nR))$ total time for h-SSSP, which is well within our $O(mh\log(n)\log(nR)/\epsilon)$ time bound. We can thus maintain $\delta'(s,v)$ in time $O(\log(n))$ time per change to $\delta'_k(s,v)$. □

The total time spent in Step 3 (see running time breakdown above) is thus $O([nh\log(nR)/\epsilon] \cdot [\text{the time to update } \delta'(s,v)]) = O([nh\log(nR)/\epsilon] \cdot [\log(n)]) = O(nh\log(n)\log(nR)/\epsilon)$, which is within our desired $O(mh\log(n)\log(nR)/\epsilon)$ time bound.

All we have left is to analyze the total time spent in Step 1 of the running time breakdown above. This will take some work. Let us focus on h-SSSP$_k$ for some particular value of k. Recall that given update increase-weight(u,v), h-SSSP$_k$ starts by scaling the weight of (u,v) in G_k. The first two steps are as follows:

- If $w_{\text{new}}(u,v) > 2^{k+1}$, h-SSSP$_k$ deletes it from G_k.
- Scale $w_{\text{new}}(u,v)$ up to the nearest multiple of $\epsilon 2^k/h$.

Thus, it is easy to see that if $w_{\text{old}}(u,v) > 2^{k+1}$ or if $w_{\text{old}}(u,v)$ and $w_{\text{new}}(u,v)$ both scale up to the same multiple of $\epsilon 2^k/h$, then there is no need to process increase-weight(u,v) in h-SSSP$_k$ as it will not affect G_k in any way. This motivates the following definitions. Recall that h is the hop-length to which we are running the h-SSSP algorithm in question.

Definition 7.6. *Let $\alpha = \epsilon/h$. Given an integer $k \in [\lfloor \log(c) \rfloor, \lceil \log(nC) \rceil]$, we say that a positive real number ζ is* k-marked *if both of the following properties hold:*

1. *ζ is an integer multiple of $2^k\alpha = \epsilon 2^k/h$.*
2. *$\zeta < 2^{k+1}$.*

We say that a number is marked *if it is k-marked for at least one value of k.*

Lemma 7.7. *Say that we are given an update increase-weight(u, v): $w_{old}(u, v) \to w_{new}(u, v)$ (an edge deletion just increases the weight to ∞). This update affects an h-SSSP$_k$ algorithm only if there is a k-marked number in the half-open interval $[w_{old}(u, v), w_{new}(u, v))$. We will refer to such an update as* crossing *the k-marked number.*

Proof. This lemma stems directly from our previous discussion. The only way for increase-weight(u, v) to change a weight in G_k is if $w_{\text{old}}(u, v) \leq 2^{k+1}$ and if $w_{\text{old}}(u, v)$ and $w_{\text{new}}(u, v)$ scale up to different multiples of $\epsilon 2^k/h = \alpha 2^k$. This is equivalent to the requirement that there is a k-marked number in the half-open interval $[w_{\text{old}}(u, v), w_{\text{new}}(u, v))$. ∎

Lemma 7.8. *For any integer $k \in [\lfloor \log(c) \rfloor, \lceil \log(nC) \rceil]$ there are $2h/\epsilon = 2/\alpha$ k-marked numbers, for a total of $2h \log(nR)/\epsilon$ marked numbers. (Recall that we are focusing on a particular h-SSSP algorithm running up to hop-length h, so h is fixed.)*

Proof. For any integer $k \in [\lfloor \log(c) \rfloor, \lceil \log(nC) \rceil]$, the k-marked numbers are all the positive multiples of $\alpha 2^k$ that are $\leq 2^{k+1}$. It is easy to see that there are exactly $2^{k+1}/(\alpha 2^k) = 2/\alpha = 2h/\epsilon$ of these. There are $\log(nC) - \log(c) = \log(nR)$ different possible values for k, yielding a total of $2h \log(nR)/\epsilon$ marked numbers. ∎

Lemma 7.9. *Given a marked number ζ, there are $O(\log(n))$ values of k for which ζ is k-marked, and we can find all of them in $O(\log(n))$ time.*

Proof. If ζ is marked, it must be k-marked for at least one k, so there must be some k such that ζ is an integer multiple of $2^k\alpha$ that is less than 2^{k+1}. In particular, for that value of k we must have $2^k\alpha \leq \zeta \leq 2^{k+1}$. Taking logs yields $k + \log(\alpha) \leq \log(\zeta)$ and $\log(\zeta) \leq k + 1$; the first inequality then yields $k \leq \log(\zeta) - \log(\alpha)$, while the second yields $\log(\zeta) - 1 \leq k$. All in all we thus have

$$\log(\zeta) - 1 \leq k \leq \log(\zeta) - \log(\alpha),$$

so we only have to consider integers k in interval $[\log(\zeta) - 1, \log(\zeta) - \log(\alpha)]$ (note that $\log(\alpha) = \log(\epsilon/h)$ is negative). This interval contains at most $1 - \log(\alpha) \leq 1 + \log(h/\epsilon) = O(\log(n))$ integers k, and for each such k we can check in $O(1)$ time if ζ is an integer multiple of $2^k\alpha$. ∎

We can now give an intuition for our algorithm. Recall from Lemma 7.7 that an update increase-weight(u, v) only needs to be processed by some h-SSSP$_k$ if it crosses a k-marked number. By Lemma 7.9 each marked number is k-marked for only $O(\log(n))$ values of k. Thus, an update increase-weight(u, v) must be processed by $O(\log(n))$ h-SSSP$_k$ algorithms for every marked number that it crosses. But since weights only increase, they go through each marked number exactly once, so by Lemma 7.8, all the updates on a single edge (u, v) go through a total of $O(h \log(nR)/\epsilon)$ marked numbers. Thus, over all weight increases on a single edge (u, v), the total number of updates to the h-SSSP$_k$ algorithms is $O(\log(n)) \cdot O(h \log(nR)/\epsilon) = O(h \log(n) \log(nR)/\epsilon)$. Thus, over all edges, the total number of times that an update affects some h-SSSP$_k$ and must be further processed is $O(mh \log(n) \log(nR)/\epsilon)$; since each h-SSSP$_k$ algorithm only needs an additional $O(1)$ time per update (the $O(\Delta)$ term in Lemma 7.2), this does not exceed our overall $O(mh \log(n) \log(nR)/\epsilon)$ time bound for h-SSSP. We now have to show that it only takes us $O(1)$ per update to determine which h-SSSP$_k$ an update should be processed in.

LEMMA 7.10. *Given an update increase-weight(u,v): $w_{old}(u,v) \to w_{new}(u,v)$, we can find the* smallest *marked number of $[w_{old}(u,v), w_{new}(u,v))$ (if it exists) in $O(1)$ time.*

Proof. By the definition of k-marked (Definition 7.6), it is clear that if there is to be a k-marked number in $[w_{\text{old}}(u,v), w_{\text{new}}(u,v))$ (for some k), then we must have that $w_{\text{old}}(u,v) \leq 2^{k+1}$. Thus, $k_0 = \lceil \log(w_{\text{old}}(u,v)) \rceil$ is the very smallest value of k for which $[w_{\text{old}}(u,v), w_{\text{new}}(u,v))$ might contain a k-marked number. Moreover, for any $k' > k_0$, an integer multiple of $2^{k'}\alpha$ is obviously an integer multiple of $2^k\alpha$. Thus, the smallest marked number of $[w_{\text{old}}(u,v), w_{\text{new}}(u,v))$ is simply $w_{\text{old}}(u,v)$ rounded up to the nearest multiple of $2^{k_0}\alpha$, which we can find in $O(1)$ time (if this nearest multiple of $w_{\text{old}}(u,v)$ is $\geq w_{\text{new}}(u,v)$, then there is no k-marked number in interval $[w_{\text{old}}(u,v), w_{\text{new}}(u,v)))$. □

The improved h-SSSP algorithm.

We now present our algorithm for processing an update increase-weight(u,v): $w_{\text{old}}(u,v) \to w_{\text{new}}(u,v)$ in such a way as to only update the relevant h-SSSP$_k$. Note that the actual processing of an update in a particular h-SSSP$_k$ is no different than before, so we simply follow the algorithm presented in Lemma 7.2. The set S will end up containing all indices k for which increase-weight(u,v) affects h-SSSP$_k$.

```
1. While(True)
    (a) Use Lemma 7.10 to find the smallest marked number $\zeta$ in $[w_{\text{old}}(u,v), w_{\text{new}}(u,v))$. If no marked number exists in this interval, terminate loop; go to Step 2.
    (b) Use Lemma 7.9 to find all $k$ for which $\zeta$ is $k$-marked, and add them to $S$.
    (c) Start over from Step 1, but this time process increase-weight$(u,v)$: $\zeta \to w_{\text{new}}(u,v)$.
2. For all $k \in S$, process the update increase-weight$(u,v)$: $w_{\text{old}}(u,v) \to w_{\text{new}}(u,v)$ in $h$-SSSP$_k$.
3. Update the $\delta'(s,x) = \min\{\delta'_k(s,x)\}$ for all affected vertices $x$.
```

Analysis. Correctness follows directly from Lemma 7.7. Let us examine the time to process a particular update increase-weight(u,v). Recall that everything written in this section is about determining which h-SSSP$_k$ are affected by the update; once we decide to process the update in a particular h-SSSP$_k$, it is processed in exactly the same way as in section 7. Thus, as in the running time breakdown above, the overall running time of our algorithm consists of three components:

1. The time spent determining which updates increase-weight(u,v) should be processed by which h-SSSP$_k$, i.e., the loop of Step 1 above.
2. The time spent actually processing the updates, i.e., Step 2 above.
3. The time spent updating $\delta'(s,v) = \min\{\delta'_k(s,v)\}$.

We showed at the beginning of this section that Steps 2 and 3 are both within our $\widetilde{O}(mh \log R/\epsilon)$ time bound (see "running time breakdown" above). We now bound the time spent on the while loop of Step 1. Running the algorithm of Lemma 7.10 in 1(a) to find a marked number only takes $O(1)$ time, but then running the algorithm of Lemma 7.9 in 1(b) takes $O(\log(n))$ time, even though in the end we may discover that the marked number ζ is in fact only k-marked for a single value of k. Thus, the running time of the loop for a single update increase-weight(u,v) is $O(1)$, plus an additional $\log(n)$ for every marked number crossed by increase-weight(u,v). Because we are in a decremental setting, all the weight increases of any particular edge (u,v) only go through each marked number once, so by Lemma 7.8, all of the different edge updates combined cross a *total* of only $O(mh\log(nR)/\epsilon)$ marked numbers, so the total spent in the while loop over *all* updates to the h-SSSP algorithm is $O(mh\log(n)\log(nR)/\epsilon + \Delta)$, as desired. We have thus proved Theorem 3.

8. Final touches.

8.1. Removing the assumption that we know R in advance. Recall from section 2 that we define c to be the lightest edge weight to appear in the graph at any point in the update sequence and C to be the heaviest such edge weight. We define $R = C/c$. Since edge weights only increase, c is just the lightest edge weight in the original graph, before any updates occur, so we know it from the start. C, however, can keep increasing, and as presented, our algorithm requires an a priori upper bound on C in order to run the right h-SSSP$_k$ algorithms: each h-SSSP algorithm consists of running an h-SSSP$_k$ algorithm for $\lfloor\log(c)\rfloor \le k \le \lceil\log(nC)\rceil$ (see Definition 7.1 and Lemma 7.3 in section 7 for a description of h-SSSP$_k$). We show in this section that we do not actually need to know C ahead of time and can instead just continually update our current bound on C.

At any point in the update sequence, define C^* to be the largest edge weight *seen so far*, and note that $C^* \le C$. Recall from section 7 that we create algorithm h-SSSP$_k$ to handle distances between 2^k and 2^{k+1}, and that h-SSSP returns $\delta'(s,v) = \min\{\delta'_k(s,v)\}$, where $\delta'_k(s,v)$ is the (s,v)-distance returned by h-SSSP$_k$. But all distances in the current graph are less than nC^*, so we have no need for h-SSSP$_k$ as long as $2^k > nC^*$; that is, for $k > \lceil\log(nC^*)\rceil$, we can think of $\delta'_k(s,v) = \infty$. Thus, instead of immediately creating h-SSSP$_k$ for all $\lfloor\log(c)\rfloor \le k \le \lceil\log(nC)\rceil$, we just create them for $\lfloor\log(c)\rfloor \le k \le \lceil\log(nC^*)\rceil$. Then, as C^* increases with updates, we start running h-SSSP$_k$ as soon as k becomes $\le \lceil\log(nC^*)\rceil$, and we add $\delta'_k(s,v)$ to the heap for $\delta'(s,v)$.

It is easy to see that the running time of this new method is no worse than if we knew C in advance and set up all the h-SSSP$_k$ from the beginning (for $\lfloor\log(c)\rfloor \le k \le \lceil\log(nC)\rceil$). It is in fact slightly faster, as we avoid processing updates that occur while $k > \lceil\log(nC^*)\rceil$.

8.2. The incremental setting. As presented, our algorithm works in the decremental setting, where we have only deletions and weight increases. However, like many other decremental algorithms, our algorithm can be made to run in the incremental setting with only the smallest of modifications. That is, it can process either a sequence of deletions/weight-increases or a sequence of insertions/weight-decreases, though certainly not a sequence of both.

Most of the description of our algorithm deals with the various graphs and shortcut edges that we construct. Yet when it comes to dynamically processing the updates, all the work is done by the various h-SSSP algorithms running on these different graphs. The h-SSSP algorithm is in turn composed of h-SSSP$_k$ algorithms. So in the end, our algorithm is merely a large collection of different h-SSSP$_k$ algorithms.

But the h-SSSP$_k$ algorithm simply runs King's algorithm for maintaining a shortest path tree, which, as has been observed in many previous papers, runs equally well in the incremental setting. This is evident from Theorem 2: the theorem itself works in the fully dynamic setting, and it is only the corollary which required a decremental setting in order to bound the total number of distance changes. This bound, however, holds equally well in the incremental setting: as long as distances only ever change in one direction (increase or decrease), the total number of changes between 1 and d is at most d.

Thus, our algorithm can be made to run in the incremental setting by simply switching all of the constituent h-SSSP$_k$ algorithms to run in the incremental setting. Everything else remains unchanged: the various graphs $G_{v,k}$, the shortcut edges, the approximation analysis, etc. There are only two minor points worth noting:

- Recall that our algorithm constructs many different graphs, and so an update in G can proliferate into multiple updates on multiple graphs. We argued that we could nonetheless run h-SSSP on these different graphs because although the update sequence differs from the perspective of each graph, it is always decremental: shortcut weights correspond to distances in the original graph, so since distances only increase, shortcut weights only increase. A symmetric claim is true of the incremental setting: since distances in the original graph only decrease, shortcut weights also only decrease, so the update sequence is incremental from the perspective of all the different graphs.
- In section 8.1 we argued that while c remains fixed, C increases over time, so our decremental algorithm kept an upper bound on C in order to have an estimate on R. In the incremental setting, it is C that is fixed and c that changes, so we instead keep a lower bound on c.

8.3. A fully dynamic algorithm. There is a standard algorithm for transforming any decremental algorithm for APSP or directed transitive closure into a fully dynamic algorithm with query-update trade-offs. The fastest update times we know how to achieve in the fully dynamic setting often stem from this technique, though at the often unacceptable cost of a polynomial query time. The technique was first introduced in a paper by Henzinger and King [9] on dynamic transitive closure and has since been used in several papers on dynamic shortest paths and dynamic transitive closure (see, e.g., [15], [14], [12]). Our application of the technique is completely identical to the one used in the cited papers, but as far as we know, none of those papers presented the result in quite sufficient enough generality for it to apply directly to our case. We thus reconstruct the technique from scratch, stating it in the most general terms possible (note that transitive closure is a special case of α-approximate APSP). Recall that $\pi(x, y)$ is the shortest path from x to y in the current version of the graph and that $\delta(x, y)$ is the length of this path.

THEOREM 5 (see [9]). *Given a decremental algorithm D for α-approximate APSP with* total *update time Λ over all updates and query time $O(q)$, for any positive integer t we can construct a fully dynamic algorithm for α-approximate APSP with* amortized *update time $O(\Lambda/t + (m + n\log(n))t)$ and query time $O(t + q)$. The fully dynamic algorithm admits the following batch updates in the same $O(\Lambda/t + (m + n\log(n))t)$ amortized time per update:*

- Batch delete. *Delete or increase the weight of an arbitrary set of edges E'.*
- Centered batch insert. *Insert or decrease the weight of a group of edges E_v that are all incident to some vertex v.*

COROLLARY 8.1. *Given our (randomized) $\widetilde{O}(mn\log R/\epsilon)$ decremental algorithm for $(1 + \epsilon)$-approximate APSP in directed weighted graphs, we can build a fully dynamic (randomized) algorithm with amortized update time $\widetilde{O}(\frac{mn\log R}{\epsilon t})$ and query time $O(t)$ for any $1 \le t \le \sqrt{n}$. Previously such a result was known only for undirected unweighted graphs.*

Proof of Theorem 5. We use the decremental algorithm D as a black box to build the desired fully dynamic algorithm. Let $D(u, v)$ be the α-approximate shortest distance from u to v returned by algorithm D.

Initialization.

- Initialize the decremental algorithm D on the starting graph G. This computes α-approximate APSP in G and maintains this information over all deletions to come.
- Create an empty list I which will contain the vertices affected by insertions.
- Create a counter C for the number of updates so far. Start with $C = 0$.

Batch-insertion(E_v).

- If $C > t$, restart the entire algorithm. That is, set $C = 0$, delete all elements from I, and reinitialize the decremental algorithm D on the current version of the graph. Else, set $C = C + 1$ and continue.
- Add v to I.
- For every vertex $x \in I$ use Dijkstra's algorithm to compute single source shortest distances to and from x.

Batch-delete(E').

- If $C > t$, restart the entire algorithm. That is, set $C = 0$, delete all elements from I, and reinitialize the decremental algorithm D on the current version of the graph. Else, set $C = C + 1$ and continue.
- Process all the deletions in the decremental algorithm D. This of course changes the various $D(u, v)$.
- For every vertex $x \in I$ use Dijkstra's algorithm to compute single source shortest distances to and from x.

Query(u, v).

- Let $I(u, v) = \min_{x\in I}(\delta(u, x) + \delta(x, v))$.
- Return $\delta'(u, v) = \min\{I(u, v), D(u, v)\}$.

Correctness proof. Note that when we compute $I(u, v) = \min_{x\in I}(\delta(u, x)+\delta(x, v))$ we know both $\delta(u, x)$ and $\delta(x, v)$ because after each update we compute shortest paths to and from each vertex in I. Thus, $I(u, v)$ contains the length of the shortest path from u to v that goes through one of the vertices in I. If $\pi(u, v)$ contains a vertex from I, our query algorithm outputs $\delta'(u, v) = I(u, v) = \delta(u, v)$. If $\pi(u, v)$ does not use any vertex in I, then the entire path $\pi(u, v)$ exists in the graph G_D, which is the graph G subjected to all the deletions in our update sequence and none of the insertions. Since the decremental algorithm D is running precisely on G_D, $D(u, v)$ is guaranteed to return an α-approximation to $\delta(u, v)$. In either case, we have that $\delta'(u, v) = \min\{I(u, v), D(u, v)\}$ is an α-approximation to $\delta(u, v)$, as desired.

Running time analysis. The fully dynamic algorithm runs for exactly t updates before restarting. Since the *total* update time of the decremental algorithm D is Λ, the amortized time for a single update is thus $O(\Lambda/t)$. Each update inserts exactly one vertex into I, so we always have $|I| \le t$. Each update also requires us to run Dijkstra's algorithm to and from each vertex in I, which takes time $O((m + n\log(n))|I|) = O((m + n\log(n))t)$. The amortized update time is thus $O(\Lambda/t + (m + n\log(n))t)$, as desired.

Each query requires us to compute $\delta(u, x) + \delta(x, v)$ for every $x \in I$. Each such value can be computed in $O(1)$ since we already know $\delta(u, x)$ and $\delta(x, v)$. The query algorithm also spends $O(q)$ time querying $D(u, v)$. The query time is thus $O(|I|+q) = O(t + q)$, as desired. ∎

9. Open problems.

Our algorithm achieves the desired $\widetilde{O}(mn/\epsilon)$ total update time in directed graphs with polynomial weights, as compared to the previous state of the art of Roditty and Zwick [14], which only achieved $\widetilde{O}(mn/\epsilon)$ for undirected unweighted graphs. We cannot really hope to beat total update time $\widetilde{O}(mn)$, as this is the running time of the fastest combinatorial algorithms for the much simpler problem of *static* APSP. But there still remain many open problems concerning decremental shortest paths.

- Can we achieve total update time $\widetilde{O}(mn)$ for decremental *exact* APSP? This would be interesting even in unweighted, undirected graphs. Currently, the best known algorithm is that of Baswana, Hariharan, and Sen [2], which achieves total update time $\widetilde{O}(n^3)$ in directed unweighted graphs.
- Bernstein and Roditty [6] showed that in undirected unweighted graphs, one can achieve total update time $O(n^{3-\epsilon})$ if one allows a constant approximation. Henzinger et al. [10] later improved upon this result; their algorithm could maintain $(2+\epsilon)$-approximate APSP in total update time $\widetilde{O}(n^{2.5})$. Can one achieve something similar for directed graphs?
- Our algorithm is the first decremental shortest paths algorithm to work on weighted graphs, but it still carries a $\log R$ factor. Very recently, Henzinger, Krinninger, and Nanongkai presented an algorithm for weighted undirected graphs that can maintain an approximate shortest path tree under deletions in near-linear total update time [11], but the $\log R$ factor remains. Is it possible to remove this? Such a result would be interesting for even the most basic possible problem of maintaining a $(1+\epsilon)$-approximate shortest path for a single $s-t$ pair in an undirected graph. Can we maintain this $s-t$ path in total update time $\widetilde{O}(mn)$ on a graph with arbitrary real edge weights?
- For *static* APSP, we can beat the $O(mn)$ bound in dense graphs by using fast matrix multiplication. Is it possible to extend these techniques to the dynamic case and achieve total update time $O(n^{3-\epsilon})$ over all deletions?
- Our decremental APSP algorithm is randomized. Very recently, Henzinger, Krinninger, and Nanongkai showed that in undirected unweighted graphs we can achieve total update time $\widetilde{O}(mn/\epsilon)$ with a deterministic algorithm [10]. Can we achieve a similar result for directed graphs, or graphs with weights polynomial in n? Note that deterministic algorithms are especially important in the dynamic setting, because they do not require the assumption of an oblivious adversary; none of the existing randomized algorithms can be used in a setting with an adaptive adversary that sees the choices made by the algorithm and adapts the update sequence accordingly.

Acknowledgment. I would like to thank my advisor Cliff Stein for a great number of helpful suggestions.

REFERENCES

[1] I.Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck, *Highway dimension, shortest paths, and provably efficient algorithms*, in Proceedings of the 21st SODA, Austin, TX, 2010, pp. 782–793.

[2] S. Baswana, R. Hariharan, and S. Sen, *Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths*, J. Algorithms, 62 (2007), pp. 74–92.

[3] A. Bernstein, *Fully dynamic approximate all-pairs shortest paths with constant query and close to linear update time*, in Proceedings of the 50th FOCS, Atlanta, GA, 2009, pp. 50–60.

[4] A. Bernstein, *Near linear time $(1+\epsilon)$-approximation for restricted shortest paths in undirected graphs*, in Proceedings of the 23rd SODA, Kyoto, Japan, 2012, pp. 189–201.

[5] A. Bernstein, *Maintaining shortest paths under deletions in weighted directed graphs (extended abstract)*, in ACM STOC, 2013, pp. 725–734.

[6] A. Bernstein and L. Roditty, *Improved dynamic algorithms for maintaining approximate shortest paths under deletions*, in Proceedings of the 22nd SODA, San Francisco, CA, 2011, pp. 1355–1365.

[7] C. Demetrescu and G. F. Italiano, *A new approach to dynamic all pairs shortest paths*, J. ACM, 51 (2004), pp. 968–992.

[8] S. Even and Y. Shiloach, *An on-line edge deletion problem*, J. ACM, 28 (1981), pp. 1–4.

[9] M. Henzinger and V. King, *Fully dynamic biconnectivity and transitive closure*, in Proceedings of the 36th FOCS, Milwaukee WI, 1995, pp. 664–672.

[10] M. Henzinger, S. Krinninger, and D. Nanongkai, *Dynamic approximate all-pairs shortest paths: Breaking the $o(mn)$ barrier and derandomization*, in IEEE FOCS 2013, 2013, pp. 538–547.

[11] M. Henzinger, S. Krinninger, and D. Nanongkai, *Decremental single-source shortest paths on undirected graphs in near-linear total update time*, in IEEE FOCS 2014, 2014, pp. 146–155.

[12] V. King, *Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs*, in IEEE FOCS, 1999, pp. 81–91.

[13] G. Ramalingam and T. Reps, *An incremental algorithm for a generalization of the shortest-path problem*, J. Algorithms, 21 (1996), pp. 267–305.

[14] L. Roditty and U. Zwick, *Dynamic approximate all-pairs shortest paths in undirected graphs*, in Proceedings of the 45th FOCS, Rome, Italy, 2004, pp. 499–508.

[15] L. Roditty and U. Zwick, *Improved dynamic reachability algorithms for directed graphs*, SIAM J. Comput., 37 (2008), pp. 1455–1471.

[16] L. Roditty and U. Zwick, *On dynamic shortest paths problems*, Algorithmica, 61 (2011), pp. 389–401.

[17] P. Sankowski, *Subquadratic algorithm for dynamic shortest distances*, in Proceedings of the 11th COCOON, Kunming, China, 2005, pp. 461–470.

[18] M. Thorup, *Compact oracles for reachability and approximate distances in planar digraphs*, J. ACM., 51 (2004), pp. 993–1024.