Fully dynamic (2 + epsilon) approximate all-pairs shortest paths with fast query and close to linear update time

2009 50th Annual IEEE Symposium on Foundations of Computer Science

Fully Dynamic $(2+\epsilon)$ Approximate All-Pairs Shortest Paths with Fast Query and Close to Linear Update Time

Aaron Bernstein*
*Massachusetts Institute of Technology
Cambridge, MA, 02139
Email: bernstei@gmail.com

Abstract— For any fixed $1 > \epsilon > 0$ we present a fully dynamic algorithm for maintaining $(2+\epsilon)$-approximate all-pairs shortest paths in undirected graphs with positive edge weights. We use a randomized (Las Vegas) update algorithm (but a deterministic query procedure), so the time given is the *expected* amortized update time.

Our query time $O(\log\log\log n)$. The update time is $\widetilde{O}(mn^{O(1/\sqrt{\log n})}\log(nR))$, where R is the ratio between the heaviest and the lightest edge weight in the graph (so $R=1$ in unweighted graphs). Unfortunately, the update time does have the drawback of a super-polynomial dependence on ϵ: it grows as $(3/\epsilon)^{\sqrt{\log n/\log(3/\epsilon)}} = n^{\sqrt{\log(3/\epsilon)/\log n}}$.

Our algorithm has a significantly faster update time than any other algorithm with sub-polynomial query time. For exact distances, the state of the art algorithm has an update time of $\widetilde{O}(n^2)$. For approximate distances, the best previous algorithm has a $O(kmn^{1/k})$ update time and returns $(2k-1)$ stretch paths. Thus, it needs an update time of $O(m\sqrt{n})$ to get close to our approximation, and it has to return $O(\sqrt{\log n})$ approximate distances to match our update time.

Keywords-graph algorithms; dynamic algorithms; approximation algorithms; shortest paths;

1. Introduction

The goal of a dynamic shortest path algorithm is to process an online sequence of *query* and *update* operations on an underlying graph. An update operation inserts or deletes an edge from the graph, while a query operation asks for the shortest distance (*in the current graph*) between two given vertices. An *approximate* dynamic algorithm only returns an approximate shortest distance between the query points. A dynamic algorithm is said to be *incremental* if it only handles edge insertions, *decremental* if it only handles edge deletions, and *fully dynamic* if it handles both.

1.1. Existing Algorithms

The efficiency of a dynamic algorithm is primarily judged by two parameters: query time and update time. Different algorithms achieve various trade-offs between these two parameters, but we only mention existing algorithms with a sub-polynomial query time.

The fastest fully dynamic *exact* algorithm was developed by Demetrescu and Italiano [1], although Thorup [5] later improved on this slightly. It works for general graphs and has an amortized update time of $\widetilde{O}(n^2)$ [1]. For *approximate* dynamic algorithms, much work has been done on the incremental and decremental cases, but it all culminated with extremely efficient algorithms developed by Roditty and Zwick [4]. They presented separate incremental and decremental algorithms which return $(1+\epsilon)$ approximate shortest paths and have a constant query time and an amortized update time of $\widetilde{O}(n)$. Both algorithms have a randomized update procedure.

However, little progress has been made for fully dynamic approximate algorithms. For unweighted graphs, Roditty and Zwick [4] presented a $(1+\epsilon)$ algorithm with an update time of $\widetilde{O}(mn/t)$ $(1 \le t \le \sqrt{m})$, but the query time is $O(t)$. Another approach is to just build the approximate distance oracle of Thorup and Zwick [6] from scratch after each update. For any k with $2 \le k \le \log n$ this leads to a $(2k-1)$ approximate algorithm with a query time of $O(k)$ and an update time of $O(kmn^{1/k})$. The algorithm works for undirected graphs with positive weights.

1.2. Our contributions

We present a dynamic approximate shortest path algorithm that works for undirected graphs with positive edge weights. Our query algorithm is deterministic, but our update procedure is randomized (Las Vegas), so we give the *expected* update time. For any fixed $\epsilon \in (0,1)$ we present an algorithm that returns $(2+\epsilon)$ shortest distances (it can also return the shortest *paths* themselves, although the query time is multiplied by the number of edges in the output path). Let $\beta = \log(3/\epsilon)$, and let m refer to the number of edges in the *current* graph.

Our query time is $O(\log\log\log n)$. In unweighted graphs, the update time is

$$O(mn^{\sqrt{\beta/\log n}}\log^{1/2} n + n\cdot n^{2\sqrt{\beta/\log n}}\log^{5/2} n) = \widetilde{O}(mn^{O(1/\sqrt{\log n})})$$

[1]We say that $f(n) = \widetilde{O}(g(n))$ if $f(n) = O(g(n)polylog(n))$

0272-5428/09 $26.00 © 2009 IEEE
DOI 10.1109/FOCS.2009.16

In weighted graphs, it is

$$O(mn^{\sqrt{\beta/\log n}}\log(nR)\log n \;+ \\ n\cdot n^{2\sqrt{\beta/\log n}}\log(nR)\log^{3/2} n) = \\ \widetilde{O}(mn^{O(1/\sqrt{\log n})}\log(nR))$$

Our update time is almost linear, but not quite: there is this extra $n^{O(1/\sqrt{\log n})}$ factor, and the dependence on ϵ is somewhat super-polynomial, so we should not set ϵ too small. However, we briefly discuss in Section 6.4 why it seems quite plausible that one could improve the update time to $\widetilde{O}(m)$ without changing the underlying structure of the algorithm.

Also, we omit the details, but our algorithm can be extended to handle updates that touch not just a single edge, but an arbitrary number of edges that are all incident on the same common vertex. The update and query times remain the same.

Note that our update time is significantly better than that of any previous algorithms with small query time. Using the approximate distance oracle of Thorup and Zwick [6] would require an update time of $O(m\sqrt{n})$ to achieve a 3-approximation. To match the update time of our algorithm, it would have to return $\Omega(\sqrt{\log n})$ approximate distances.

1.3. Organization

Section 2 introduces basic preliminaries. Section 3 presents an algorithm Small-Diam that can efficiently find "short" shortest paths in weighted graphs. That is, it only guarantees a good approximation if the actual shortest distance is small. Like many approximate algorithms, Small-Diam has each vertex store its own *local* distance information, with some storing more than others. But in our case, low information vertices do not store information about less of the graph: rather, they store *full* distance information but in an *outdated* version of the graph (*i.e* a version that has since been changed by updates). The query algorithm patches together information from these various versions to find an approximate shortest path. During updates, we *renew* several vertices by updating their local version of the graph to the current version. The update procedure of Small-Diam is described in section 3 but generalized in section 5.2.

Section 4 extends Small-Diam to efficiently find shortest paths with few edges; in weighted graphs, this is more powerful than finding short shortest paths. We do this by modifying the edge weights of our graph in such a way that algorithm Small-Diam *as is* runs more quickly. As long as a path has few edges, these weight changes do not alter its length by much.

Section 5.1 gives intuition for how to proceed. We want to run Small-Diam, so we need to ensure that all shortest paths in our graph have few edges. We do this by adding heavy edges that do not change shortest distances, but which ensure that an approximate shortest path can always be patched together from just a few of these edges. More formally, we create an *emulator* H for our graph: H has the same vertex set and similar shortest distances but uses different edges.

Sections 6.1 and 6.2 present the emulator H that we use, while section 6.3 shows how we can efficiently maintain this emulator during changes to our underlying graph. The techniques in these sections rely heavily on those used by the approximate distance oracles of Thorup and Zwick [6].

2. NOTATION

Let $G = (V, E)$ be our *undirected* graph with positive edge weights, and let $w(u, v)$ be the weight of edge (u, v). For any pair $x, y \in V$ let $\pi(x, y)$ be the shortest $x - y$ path, and let $d(x, y)$ be the weight of $\pi(x, y)$. We assume that the minimum edge weight in the graph is 1; we can ensure this by multiplying all the weights by the same number. We let R be the maximum edge weight in the resulting graph (so R is the ratio of the heaviest to the lightest edge weight in the original graph).

Our algorithm can handle updates that insert a single edge of arbitrary weight, delete a single edge, or change the weight of an existing edge. For simplicity, we will not deal with weight changes because they can always be represented as a deletion followed by an insertion. A query algorithm takes as input a pair of vertices x, y and outputs an approximation to $d(x, y)$.

We say that an algorithm outputs an α approximation if given any query input x, y it outputs a value in $[d(x, y), \alpha d(x, y)]$. Throughout the paper, ϵ refers to an arbitrary positive constant < 1. Note that for any constant c we can treat $c\epsilon$ as ϵ and $(1 + \epsilon)^c$ as $(1 + \epsilon)$ because we can just replace ϵ with $\epsilon' = \epsilon/2c$.

2.1. k-algorithms

Given a query x, y, our algorithm runs differently depending on $d(x, y)$. This may seem circular because we do not know $d(x, y)$ ahead of time, but our solution is to just guess several different values for $d(x, y)$. In particular, for the rest of this paper, we will only focus on queries x, y for which $k \leq d(x, y) \leq (1 + \epsilon)k$, where k is an arbitrary parameter. Intuitively, this is a safe assumption because our interval sizes increase exponentially by powers of $(1 + \epsilon)$, so there are few relevant values of k to check.

Thus, the rest of this paper will focus on showing that for *any* k we can develop a *k-algorithm* which satisfies all of the following properties:

- the query time of the k-algorithm is constant.

• if $d(x,y) \leq (1+\epsilon)k$ then the algorithm returns $\delta(x,y)$ satisfying $d(x,y) \leq \delta(x,y) \leq (2+\epsilon)(1+\epsilon)k$.
• if $d(x,y) > (1+\epsilon)k$ then the algorithm returns $\delta(x,y)$ satisfying $\delta(x,y) \geq d(x,y)$.

This definition of a k-algorithm is slightly complicated, but the basic idea is that the k-algorithm returns a $(2+\epsilon)(1+\epsilon) < (2+4\epsilon)$ approximation to $d(x,y)$ as long as $d(x,y) \in [k,(1+\epsilon)k]$. Note that additive errors of ϵk are allowed in a k-algorithm because they are equivalent to $(1+\epsilon)$ multiplicative errors.

During updates, we update every k-algorithm for $k = 1,(1+\epsilon),(1+\epsilon)^2,(1+\epsilon)^3...nR$. Our query procedure queries in each of these k-algorithms and then takes the minimum of the results. This yields a $(2+\epsilon)(1+\epsilon)$ approximation because bullets 2 and 3 ensure that all values in the min clause are $\geq d(x,y)$, and the second bullet ensures that one of the values is a good approximation.

There are $O(\log(nR))$ different values of k to check so that is the query time. We can improve this to $O(\log\log(nR))$ by binary searching for the right value of k instead of naively checking each one. In fact, we can further improve to $O(\log\log\log n)$ by using an $O(\log n)$ approximate distance oracle of Thorup and Zwick [6] to start with a $\log n$ approximation, thus allowing us to narrow our search space (details omitted). *Thus, from now on, we only focus on presenting a k-algorithm.*

3. An Algorithm for Small Diameter Graphs

This section describes an algorithm Small-Diam for efficiently finding "short" shortest paths in integer-weighted graphs. That is, given some parameter $d \leq n$, the output $\delta(x,y)$ is guaranteed to be a $(2+\epsilon)$-approximation to $d(x,y)$ *as long as* $d(x,y) \leq d$ (technical note: G itself may not have integer weights, but we later modify its edge-weights to change this). Otherwise, the only guarantee is that $\delta(x,y) \geq d(x,y)$. The (amortized) update time is $O(md)$, which is good for small d. Note that Small-Diam serves as a k-algorithm with update time $O(mk)$, but that it is slightly more general than a k-algorithm because it only requires that we bound $d(x,y)$ from *above*.

3.1. An Existing Technique

We rely on an existing algorithm for decrementally maintaining a single shortest path tree. The algorithm of Even and Shiloach [2] is restricted to undirected, unweighted graphs, but King [3] extended this to weighted, directed graphs.

King's algorithm (see section 2.1 of [3]) assumes a graph with integer edge weights in a decremental setting. Given a source s and a distance d it maintains a shortest path tree from s *up to distance* d. The update time over all deletions is $O(md)$.

The intuition behind King's algorithm is that when we delete an edge, we only have to worry about vertices whose distance from s changed. In particular, the data structure only explores the edges incident on a vertex x when the shortest distance from s to x increases. But we are only maintaining a shortest path tree up to distance d, so since the graph has integer edge weights, the distance from s to x can increase at most d times before x leaves our jurisdiction. Note that a decremental setting ensures that distances only increase, never decrease.

Definition 3.1. We refer to the above data structure from a source s up to distance d as $EDS(s,d)$ (EDS stands for exact decremental structure).

3.2. Local Information

The algorithm above provides a method for handling deletions. In particular, since this section only handles shortest paths of length $\leq d$, we construct an $EDS(s,2d)$ from each vertex s (we need $2d$ not d because we are returning an approximation). We process deletions in G by updating every EDS. However, $EDS's$ cannot handle insertions, so we largely ignore insertions to G. This means that our vertices will store shortest path trees in *outdated* versions of the graph: they might not know about recently inserted edges.

To handle insertions, we occasionally *renew* certain vertices s by building a new $EDS(s,2d)$. This ensures that s knows about all edges that existed at the time of the renewal. The new $EDS(s,2d)$ now acts as before: it processes all further deletions to G but ignores insertions.

To keep track of the various graph versions we use a notion of *time*: we start at time 0, and then increment the time counter with each update to G.

Definition 3.2. Let $T(u,v)$ refer to the time at which edge (u,v) was most recently inserted. Let $T(u)$, the time of a vertex, be the maximum of the times of its incident edges.

Definition 3.3. We *renew* a vertex s by rebuilding $EDS(s,2d)$. Let $I(s)$ be the time at which s was most recently renewed (I stands for information).

Definition 3.4. G always refers to the current graph. Let $G_s = (V,E_s)$ refer to the version of the graph used by $EDS(s,2d)$. In particular, $E_s = \{(u,v) \in E \mid T(u,v) \leq I(s)\}$. Let $d_s(s,v)$ refer to the shortest distance between s and v in G_s. Note that we always have $E_s \subseteq E$ and $d_s(s,v) \leq d(s,v)$, which makes sense because $EDS(s,2d)$ processes all deletions.

3.3. Update Procedure

We now present a strategy for renewing vertices. For the sake of intuition, say that every EDS is fully updated, and that we insert a new edge (u, v). Then, for any pair x, y there are two possible cases. If $\pi(x, y)$ does not use (u, v) then $d(x, y) = d_x(x, y)$ so we are set. Otherwise, we have $d(x, y) = d(x, u) + d(u, y)$, in which case it would suffice to know shortest distances from u.

This suggests a simple update strategy: whenever we insert an edge (u, v) we renew both u and v. So overall, we begin by building $EDS(s, 2d)$ for every vertex s. Then, as the algorithm progresses, we maintain the $EDS's$ under deletions, and renew 2 vertices for every insertion. By Section 3.1, maintaining an EDS *over all future deletions* takes a total of $O(md)$ time. Thus, our amortized update time is $O(md)$ because every insertion builds 2 EDS structures.

(Note that with this update procedure we always have $I(v) = T(v)$. This changes in Section 5.2)

3.4. Queries

Intuitively, if we need to find $d(x, y)$, we can start by looking at $d_x(x, y)$. But $EDS(x, 2d)$ might be outdated so we also need to check the distance through every edge that was inserted after time $I(x)$. Unfortunately, checking every edge might take too long. We overcome this by associating each vertex x with a nearby high-information vertex w. To find distances from x we first take the path to w, and then rely on $w's$ high information to get to the destination.

A naive approach would be to always route through the highest information vertex in the graph, but this vertex might be too far away. Instead, we note that since our goal is a multiplicative approximation, how long of a detour we can afford depends on how far away y is. If $d(x, y)$ is small then we can only afford a short detour, but the hope is that in this case, since y is nearby, we do not need the vertex w to have all that much information.

Thus, each vertex x stores the highest information vertex among *nearby* vertices, among *medium distance* vertices, and so on. In particular, each x stores the following heaps $H(x, i)$:

$$H(x, i) = \{v | \min\{d_x(x, v), d_v(v, x)\} \leq (1 + \epsilon)^i\}$$

Note that there are only $O(\log d) = O(\log n)$ heaps per vertex x (we chose $d \leq n$). We make each $H(x, i)$ a max-heap, where vertex v has key $I(v)$. Thus, we can efficiently keep track of the highest information vertex in each heap. Since the heaps only rely on the information in the $EDS's$ we can easily maintain them along with the $EDS's$ without increasing our update time by more than a polylog factor.

Query Algorithm: Given input x, y, we try detouring through the highest information vertex in each $H(x, i)$ and each $H(y, i)$, and then take the minimum resulting distance. Letting v_i, w_i be the highest information vertices in $H(x, i), H(y, i)$ respectively, we output

$$\delta(x, y) = \min_i\{d_x(x, y), d_y(x, y), d_{v_i}(v_i, x) + d_{v_i}(v_i, y), d_{w_i}(w_i, y) + d_{w_i}(w_i, x)\}$$

There are $O(\log d)$ values of i, so the query time is $O(\log d)$. In fact, if all we need is a k-algorithm then we can reduce the query time to constant because we only need to check heaps $H(x, 1 + \left\lceil \log_{(1+\epsilon)}(k/2) \right\rceil)$ and $H(y, 1 + \left\lceil \log_{(1+\epsilon)}(k/2) \right\rceil)$ – details omitted.

Theorem 3.1. *In the above algorithm, the output* $\delta(x, y)$ *is in* $[d(x, y), (2 + \epsilon)d(x, y)]$.

Proof: See Figure 1. Every term in the min clause is $\geq d(x, y)$, so we need to show that at least one of the terms is $\leq (2 + \epsilon)d(x, y)$. Let (u, v) be the most recently inserted edge on $\pi(x, y)$, and WLOG, say that $d(x, u) \leq d(x, y)/2$. Note that when u was renewed due to the insertion of (u, v) all the other edges in $\pi(x, y)$ were already in place, so G_u contains all of $\pi(x, y)$. Thus, $d(x, y) = d_u(u, x) + d_u(u, y)$.

Now, let $H(x, i)$ be the smallest heap (minimum i) that contains u. If u is the highest information vertex in $H(x, i)$ then $u = v_i$, so the query algorithm's min clause will contain $d_u(u, x) + d_u(u, y) = d(x, y)$, as desired. Otherwise, say that w is the highest information vertex in $H(x, i)$. Note that since $I(w) \geq I(u)$, $G_w \supseteq G_u$ must also contain all of $\pi(x, y)$.

We now show that $d_w(w, x) + d_w(w, y) \leq (2 + \epsilon)d(x, y)$. For note that since $w \in H(x, i)$ and $u \notin H(x, i - 1)$ we must have that $d_w(w, x) \leq (1 + \epsilon)d_u(u, x)$. Moreover, since G_w contains $\pi(x, y)$ we have $d_w(w, y) \leq d_w(w, x) + d(x, y)$. This yields

$$d_w(w, x) + d_w(w, y) \leq (2 + 2\epsilon)d(u, x) + d(x, y) \leq (2 + 2\epsilon)(d(x, y)/2) + d(x, y) \leq (2 + \epsilon)d(x, y)$$

(Note that $d_w(w, x)$ and $d_w(w, y)$ are both $< 2d(x, y) \leq 2d$, so using $EDS(w, 2d)$ is enough). ■

4. Weighted Versus Unweighted Diameter

For this section, we focus on describing a k-algorithm for some specific k (see Section 2.1).

The update time of Small-Diam stems from the fact that it takes $O(md)$ to build and maintain a single $EDS(s, d)$. But our EDS maintains *exact* distances from a source, whereas we are willing to settle for $(1 + \epsilon)$ approximations. Thus, instead of exploring a vertex x *every* time its distance from s changes, we would like to only explore x when the distance changes

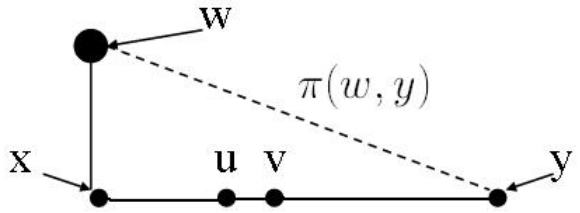

Figure 1. Proof for query algorithm of Small-Diam. (u, v) is the most recently inserted edge on $\pi(x, y)$, so u and v know about all of $\pi(x, y)$. However, there might be a *higher* information vertex w in the same heap as u, in which case we end up taking the path $\pi(x, w) \circ \pi(w, y)$

by a significant factor.

In particular, we will ensure that the distance always changes by at least some additive factor β. We can do this by rounding all of our edge weights *up* to the nearest multiple of β. Of course, the problem is that additive factors change the shortest path structure. But recall that we are focusing on a k-algorithm, so since we only need a $(1+\epsilon)$ approximation, we can afford an additive error of $k\epsilon$. Thus, we can set β to be relatively high *as long as the shortest path from s to x contains few edges, so the additive factor does not have time to grow much*. In fact, it is enough for some *approximate* shortest path from s to x to contain few edges.

Definition 4.1. We say that a graph G has an AUD_k of d if for every pair x, y with $d(x, y) \leq 4k$ there exists a $(1+\epsilon)$-approximate shortest path between x and y that contains at most d edges (AUD stands for approximate unweighted diameter, and the k is for k-algorithm).

Theorem 4.1. *As long as* **at all times** *G has an AUD_k of at most d, algorithm Small-Diam can run as a k-algorithm with an update time of only $O(md)$ (as opposed to $O(mk)$). The new approximation ratio is $(2+\epsilon)(1+\epsilon)^3$, which is within our limits.*

Remark. Theorem 4.1 is crucial because it allows Small-Diam to efficiently handle weighty paths with few edges. Our goal is now to turn G into a graph with small AUD_k; while there is no way to avoid that G might have weighty paths, we can try to add some heavy edges which do not decrease shortest distances, but which do ensure that an approximate shortest path can always be patched together from just a few of these added edges.

Proof: Recall that we are focusing on a specific k-algorithm. We start by creating a new graph G'_k that has the same edges as G, but with different weights. In particular, we round every edge weight in G *up* to the nearest multiple of $\lfloor \epsilon k/d \rfloor$. We then run Small-Diam on G'_k instead of G, so given a query (x, y), our k-algorithm for G returns a value $\delta'(x, y)$ with $d'(x, y) \leq \delta'(x, y) \leq (2+\epsilon)d'(x, y)$ ($d'(x, y)$ is the shortest $x - y$ distance in G'_k). We now need to prove that $\delta'(x, y)$ satisfies the requirements of a k-algorithm for G (see Section 2.1), and that the update time of Small-Diam in G'_k is only $O(md)$ (rather than $O(mk)$).

(technical note: the naive way to run Small-Diam on G'_k requires running $O(\log(nR))$ k'-algorithms – this multiplies our overall update time by a factor of $O(\log(nR))$ and the query time by $O(\log\log\log n)$. However, a slightly more careful analysis shows that only the values $k' = k, (1+\epsilon)k, (1+\epsilon)^2 k$, and $(1+\epsilon)^3 k$ are relevant, which allows us to avoid this increase (details omitted)).

Note that $\delta'(x, y)$ trivially satisfies property 3 of k-algorithms (because $d(x, y) \leq d'(x, y) \leq \delta'(x, y)$), so we only need to prove property 2. Thus, we can assume that $d(x, y) \leq (1+\epsilon)k$. By definition of AUD_k, there exists some $x - y$ path $P(x, y)$ in G such that $P(x, y)$ contains at most d edges and $w(P(x, y)) \leq (1+\epsilon)d(x, y)$. Let $P'(x, y)$ be the corresponding path in G'_k. Now, note that every edge weight in G'_k is larger than the corresponding edge weight in G by an additive factor of at most $\lfloor \epsilon k/d \rfloor$. Thus, since $P(x, y)$ contains at most d edges we have

$$d'(x, y) \leq w(P'(x, y)) \leq w(P(x, y)) + d\lfloor \epsilon k/d \rfloor \leq$$
$$(1+\epsilon)d(x, y) + \epsilon k \leq (1+\epsilon)^2 k + \epsilon k \leq (1+\epsilon)^3 k$$

which implies

$$\delta'(x, y) \leq (2+\epsilon)d'(x, y) \leq (2+\epsilon)(1+\epsilon)^3 k$$

Thus, bullet 2 of k-algorithms is in fact satisfied for G.

Recall that the update time of Small-Diam in G'_k is simply the amount of time it takes to maintain an $EDS(s, 4k)$ (in G'_k) over all deletions. Now, note that all edge weights in G'_k are multiples of $\lfloor \epsilon k/d \rfloor$; we consider the integer-weighted graph G''_k that is obtained by dividing every edge weight in G'_k by $\lfloor \epsilon k/d \rfloor$. G''_k has the same shortest path structure as G'_k, so maintaining an $EDS(s, 4k)$ in G'_k is equivalent to maintaining an $EDS(s, 4k/\lfloor \epsilon k/d \rfloor) = EDS(s, O(d/\epsilon))$ in G''_k, which we know takes a total of $O(md/\epsilon)$ time.

(perhaps a more intuitive proof is to recall that in maintaining $EDS(s, 4k)$, we only explore a vertex x (and incident edges) when the distance from s to x changes. But note that distances in G'_k can never change by less than $\lfloor \epsilon k/d \rfloor$, so the distance from s to x can change at most $4k/\lfloor \epsilon k/d \rfloor = O(d/\epsilon)$ times before x leaves our jurisdiction. ∎

5. Ensuring a Small AUD_k

5.1. A Suitable Emulator

Definition 5.1. We say that graph $H = (V, E')$ is an α-*emulator* for our graph G if H has the same vertex set

as G, and if for any pair of vertices x, y, we have that $d(x,y) \leq d_H(x,y) \leq \alpha d(x,y)$ (d_H refers to shortest distances in H).

By Theorem 4.1, we are set if G has a small AUD_k. Of course, this may not be the case, so our solution is to create a $(1+\epsilon)$ emulator H that has small AUD_k, and then run Small-Diam on H.

That is, every time we update an edge in G, we make corresponding changes to H that preserve its small AUD_k and maintain its $(1+\epsilon)$ emulator properties. Algorithm Small-Diam runs on H, so it does all the appropriate steps (renew vertices, update heaps, etc.) every time an edge in H is updated. By Theorem 3.1, the algorithm returns $(2+\epsilon)$ approximate distances in H, which are guaranteed to be $(2+\epsilon)(1+\epsilon) < (2+4\epsilon)$ approximate distances in G.

The only catch is that although updates in H are fast (because H has small AUD_k), a single update in G may force us to make many changes to H (to preserve the emulator properties). Thus, we need to ensure that H is easily maintainable. Ideally, every update in G would only lead to a constant number of updates in H, but this is difficult to ensure. Instead, we rely on the fact that many emulators are *localized*: an update to edge (u,v) in G only affects edges that are "near" u and v in H. So even though we might have to change many edges in H, they will all be in a small area.

5.2. Flexible Updates in the Small-Diam Algorithm

Definition 5.2. A *bulk* update is one in which many edge changes occur at once. All of the touched edges are considered to have been updated at the same time.

This notion captures the intuition above: every time we update G, we must do a bulk update on H to maintain all the required properties. Our previous approach would require algorithm Small-Diam to renew all endpoints of the updated edges, which we cannot afford to do. Thus, we rely on the intuition that our bulk updates will typically only affect nearby edges.

Recall that our reason for renewing u and v when we inserted edge (u,v) was to ensure that if some shortest path needed (u,v), then u would have enough information to handle this. But intuitively, if some vertex w near u is already being renewed, then there is no need to renew u, as all paths that use edge (u,v) can just be routed through w. As long as $d(u,w)$ is small enough, this additional detour will fit within our approximation limits.

We now describe this more formally as a property that needs to be maintained *at all times*. Recall that we are focusing on a k-algorithm (see Section 2.1), so we can assume that our shortest distances are in $[k, (1+\epsilon)k]$.

Information Property: Given any vertex u, there must exist a vertex w for which the following 2 properties hold. We refer to w as the *hub* of u.
1) $d(u,w) \leq \epsilon k$
2) $I(w) \geq T(u)$ (recall Definitions 3.2 and 3.3)

Theorem 5.1. *As long as property 1 is maintained, the query algorithm of Small-Diam returns $(2+\epsilon)$-approximate shortest paths.*

Remark. This theorem implies that we can be much more flexible with our renewal strategy for bulk updates. All we have to do is find a small set of vertices whose renewal preserves property 1.

Proof: The proof is almost identical to that of Theorem 3.1. As before, given an input pair x, y, we look at the edge (u,v) on $\pi(x,y)$ that was most recently inserted. Previously, we relied on the fact that u was renewed after the insertion of (u,v), so u knew about all of $\pi(x,y)$. But with property 1, even if u itself does not have the necessary information, there must be a nearby hub w that does. Thus, the proof continues exactly as before, just with w instead of u. The added distance between u and w is at most ϵk, which is insignificant for a k-algorithm (see Section 2.1). ■

6. The Emulator

We now describe an emulator with small AUD_k (Definition 4.1). The emulator we use is almost identical to a spanner of Thorup and Zwick [7], although we use it for very different purposes. It is based upon techniques of the same authors that were originally developed for approximate distance oracles [6].

6.1. The Techniques of Thorup and Zwick

In their spanner, Thorup and Zwick let different vertices have different *priorities*: high priority vertices are rarer, but they are also more well connected. To get from one vertex to another, we take paths to get to higher and higher priority vertices, which get us closer and closer to our destination.

Definition 6.1. Let $V = A_0 \supseteq A_1 \supseteq ... \supseteq A_{c-1} \supseteq A_c = \emptyset$ be sets of vertices (c is a parameter of our choosing). We say that a vertex has *priority* i, or is an *i-vertex* if it is in A_i/A_{i+1}. We define $d(v, A_i)$ to be the shortest distance from v to some i-vertex: $d(v, A_i) = \min_{w \in A_i/A_{i+1}} d(v,w)$.

Definition 6.2. Given an i-vertex v, we define the *cluster* of v to be $C(v) = \{w \in V \mid d(w,v) < d(w, A_{i+1})\}$

We use the same sets A_i as Thorup and Zwick [6]. In particular, we start with $A_0 = V$, and every vertex in

A_i is independently sampled and put into A_{i+1} with probability $1/n^{1/c}$. This leads to:

Lemma 6.1. *[6] The expected total size of all the clusters is $O(cn^{1+1/c})$.*

6.2. Our Emulator

In their spanner, Thorup and Zwick [7] connected each vertex v to all the vertices in $C(v)$. But recall that in order for our emulator to be easily maintainable (as G changes), we need it to be "localized": vertex v should only be connected to nearby vertices.

Definition 6.3. Given a parameter p, we define the *p-truncated* cluster of v to be

$$C_p(v) = \{w \mid w \in C(v) \text{ and } d(v,w) \leq p\}$$

To keep our emulator localized, we only connect v to vertices in the truncated cluster of v. To determine the appropriate truncation distance, we recall that we are developing a k-algorithm, so we can afford additive errors of ϵk. Thus, for every vertex v and every $w \in C_{\epsilon k/2}(v)$ we add an edge (v,w) of weight $d(v,w)$ to our emulator. *We also add all the edges of the original graph G to our emulator*. Let H be the resulting emulator.

Theorem 6.2. *[7] In expectation, the above emulator H has $O(m + cn^{1+1/c})$ edges and takes $O(cmn^{1/c})$ time to construct. Also, H is a 1-emulator (shortest distances in H are identical to those in G).*

Proof: For the non-truncated version of H, the bound on the number of edges stems from Lemma 6.1, and the construction algorithm is described in [6]. It is easy to check that truncation only decreases these parameters. H is a 1-emulator because all edges from G are included in H. ■

Theorem 6.3. *the AUD_k (see Definition 4.1) of H is $O((3/\epsilon)^c \log n)$, where c is the number of vertex priorities (the AUD_k is actually $O([(2+\epsilon)/\epsilon]^c \log n)$, but we prefer to keep notation simple). See section 7 for the proof.*

6.3. Maintaining the Emulator

All we have left to show is how to maintain the emulator H. Recall that every update in G leads to a bulk update in H. In particular, an update in G may change the clusters around various vertices in H, so we need to add and delete edges in H to ensure that we continue to have an edge from every v to every $w \in C_{\epsilon k/2}(v)$ (H must also contain all the edges of G but this is trivial to maintain, so we only focus on cluster edges).

We determine the edges that need to be updated in H by recomputing all the clusters from scratch every time G is updated: by Theorem 6.2 this only takes $O(cmn^{1/c})$ time in expectation. This leads to a bulk update with all the necessary changes to H. We now need to determine a renewal strategy for this bulk update (see Theorem 5.1).

Theorem 6.4. *If we insert or delete edge (u,v) in G and make the corresponding bulk update to H, then renewing u and v in H maintains the information property from Section 5.2 (for k-algorithms).*

Proof: The theorem relies on the fact that we use truncated clusters, so all vertices in a single cluster are near each other. There are two ways the information property might be broken, so we now show that renewing u and v takes care of both:
1: If some vertex w used edge (u,v) to get to its hub h, then deleting (u,v) makes this path unavailable. But note that if (u,v) was on $\pi(w,h)$ then u is even closer to w than h was, so we can make u the new hub of w.
2: If some edge (w,w') is inserted into H (as a result of edge (u,v) being changed in G) then $T(w)$ and $T(w')$ go up so we need to ensure that either u or v serve as hubs for w and w'. To do this, we explore the reasons why (w,w') might have been inserted into H.

The first possibility is that inserting (u,v) decreased $d(w,w')$ (in G), so w' is now in $C_{\epsilon k/2}(w)$. But then $d(w,u)$ and $d(u,w')$ are both smaller than $d(w,w')$, which in turn is $\leq \epsilon k/2$ (by the definition of truncated clusters). Thus, u is close enough to w and w' to become their new hub.

The second possibility is that deleting (u,v) *increased* $d(w,A_{i+1})$ (where i is the level of w'), resulting in w becoming part of $C_{\epsilon k/2}(w')$ (see Definition 6.1). But in this case, $d(w,u)$ must be smaller than the distance $d(w,A_{i+1})$ was *before* (u,v) *was deleted*. Also, we know that $d(w,w')$ was greater than $d(w,A_{i+1})$ before the deletion (otherwise, w would have been part of $C_{\epsilon k/2}(w')$). Thus, after the deletion we have $d(w,u) < d(w,w') \leq \epsilon k/2$, so $d(u,w') \leq d(u,w) + d(w,w') < \epsilon k$. Hence, u is close enough to both w and w' to become their new hub. ■

6.4. Putting it All Together

Remark. We have described our approach as running algorithm Small-Diam on the emulator H. But technically, we actually run Small-Diam on a slight modification of H. For recall from Theorem 4.1 that we make Small-Diam run quickly on graphs with small AUD_k (see Definition 4.1) by modifying the edge weights of the graph. Thus, what we actually have is our emulator H and a modification H': H' contains the same edges as H, just with different weights. When we update G, we compute the necessary changes to H, then we modify the edge weights of H to get to H', and only then do we run Small-Diam (renew vertices, keep track of

heaps, etc.) on H'. Since distances in H' are similar to those in H (see Theorem 4.1), it is easy to check that Theorem 6.4 guarantees that the information property is also preserved in H' (not just in H).

Update Time Analysis: When we update G, we have to recompute the cluster structure of G (to maintain H), and we need to renew two vertices in H. By Theorems 6.2 and 6.3 it takes $O(cmn^{1/c})$ to compute the cluster structures, and H has $O(m+cn^{1+1/c})$ edges, and an AUD_k of $O((3/\epsilon)^c \log n)$. Thus, by Theorem 4.1, the amortized update time *for a single k-algorithm* (there are $O(\log(nR))$ of them) is $\widetilde{O}(cmn^{1/c} + (m + n^{1+1/c})(3/\epsilon)^c \log n)$. Setting $c = \sqrt{\log(n)/\log(3/\epsilon)}$, we get the update time of Section 1.2.

We can achieve a slightly faster update time in unweighted graphs by reducing the number of edges in the emulator to $O(n^{1+1/c})$. We do this by simply not including the original edges of G in H. This leads to some complications as shortest distance in H are no longer *identical* to those in G, they are only *similar*. We omit the details but we can overcome this by slightly tweaking our algorithm and analysis; all of the basic ideas and techniques remain exactly the same. See Section 1.2 for the resulting update time.

Remark. Perhaps the easiest way to improve the update time of our algorithm would be to develop an emulator that works better for our purposes. We need a $(1+\epsilon)$ emulator that is sparse, has small AUD, and is easy to maintain as G changes. The spanner of Thorup and Zwick [7] serves us relatively well, but it was not intended to have small AUD, so it seems likely that we could do better if we developed a new emulator with this specific problem in mind.

7. Proof of Theorem 6.3

See Section 6.1 for a list of important definitions, and recall that we are focusing on a k-algorithm. To recap, our emulator H adds an edge of weight $d(v,w)$ from every vertex v to every $w \in C_{k\epsilon/2}(v)$. We want to prove the following:

Theorem 7.1. *the AUD_k (see Definition 4.1) of H is $O((3/\epsilon)^c \log n)$, where c is the number of vertex priorities (the AUD_k is actually $O([(2+\epsilon)/\epsilon]^c \log n)$, but we prefer to keep notation simple).*

Proof: Let $d = (3/\epsilon)^c$. We first prove the theorem for non-truncated clusters, and later show that this is enough. Also, we only prove the bound $AUD_k = O(cd \log n)$ (almost as good because $c = O(\log n)$). Reducing this to $O(d \log n)$ requires a slightly more careful analysis which we omit.

Let us focus on a specific pair of vertices x, y. We need to show that there exists a $(1+\epsilon)$ shortest $x-y$ path that has $O(cd \log n)$ edges. If $\pi(x,y)$ has fewer than d edges then the statement trivially holds because H contains all edges in G.

Otherwise, our approach is to show that with only $O(c)$ edges we can find a $(1+\epsilon)$ approximate path to some x' on $\pi(x,y)$ for which $d(x,x') \geq \lfloor d(x,y)/d \rfloor$. We then repeat from x' (instead of x). At each step, we get about a $1/d$ fraction of the way to y, so it is clear that we will need to take at most $O(d \log n)$ such steps. Each step uses at most $O(c)$ edges, which yields a bound of $O(cd \log n)$ for AUD_k.

(a more careful analysis here would allow us to shave the factor of c off our bound for AUD_k. The basic intuition is that the more edges we use in a step, the closer we get to y. So if we were to actually use c edges in a single step then we would get much further than a $1/d$ fraction of the way to y, so we would not have to take as many steps).

Definition 7.1. Let x_r refer to the vertex on $\pi(x,y)$ that is at distance r from x (so x_r only exists for some values of r). Also, let $\alpha = \lfloor d(x,y)/d \rfloor$. For simplicity, we let x_α be the first vertex on $\pi(x,y)$ that is at distance *at least* α from x.

Our goal is to find a $(1+\epsilon)$ shortest path with $O(c)$ edges to some vertex past x_α. The intuition for our approach is as follows. Let x' be the furthest vertex on $\pi(x,y)$ that is still in $C(x)$. If x' is past x_α then we just take the cluster edge (x,x') and we are done. Otherwise, by definition of $C(x)$, there must be some higher priority priority vertex c near x'. Our approach is to take the relatively short path $(x,x') \circ (x',c)$ and continue from c; this deviates from the original path $\pi(x,y)$ but we can afford small detours. Now, either we can get past x_α from c, or c is relatively near an even higher priority vertex c_2.

Continuing in this fashion, at each step, either we make a lot of progress toward y or we get to a nearby higher priority vertex. The higher the vertex priority, the more information is stored, and so the more progress we will eventually make toward y. There are c vertex priorities so the resulting path only uses $O(c)$ edges.

More formally, Let $p'(1)$ be the index such that $x_{p'(1)}$ (see Definition 7.1) is the furthest vertex on $\pi(x,y)$ that is still in $C(x)$. Let $x_{p(1)}$ be the vertex succeeding $x_{p'(1)}$ on $\pi(x,y)$. If $p(1) \geq \alpha$ (see Definition 7.1 for α) then we just take the path $(x, x_{p'(1)}) \circ (x_{p'(1)}, x_{p(1)})$ in H and we are done. If $p(1) < \alpha$ then by definition of $C(x)$ there must be some 1-vertex v_1 such that $d(x_{p(1)}, v_1) \leq d(x_{p(1)}, x) = p(1) < \alpha$. Thus, we take the path $(x, x_{p'(1)}) \circ (x_{p'(1)}, x_{p(1)}) \circ (x_{p(1)}, v_1)$, and proceed from the higher priority vertex v_1.

Let $x_{p'(2)}$ be the furthest vertex on $\pi(x,y)$ that is still in $C(v_1)$, let $x_{p(2)}$ be the vertex succeeding it on $\pi(x,y)$, and let v_2 be the 2-vertex that is closest to

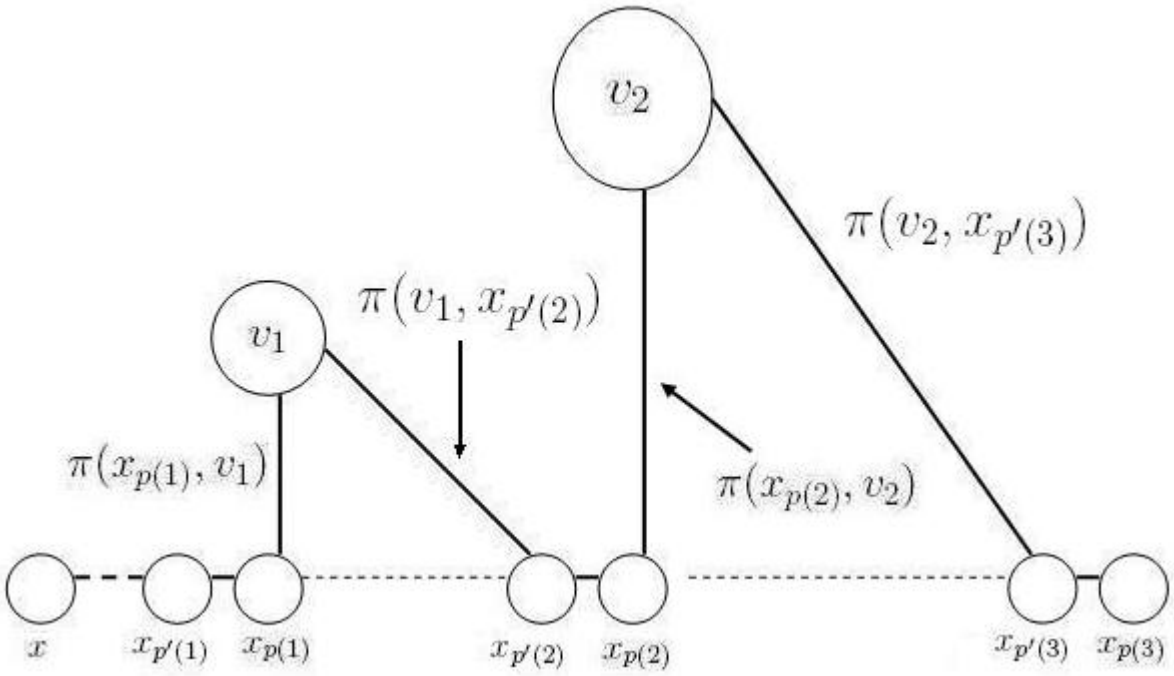

Figure 2. This figure shows our path from x to $x_{p(i)}$ in H that contains few edges. The dotted path is the actual shortest path from x to $x_{p(i)}$, while the bold path is the one we use.

$x_{p(2)}$. Let $x_{p'(3)}$ be the vertex furthest on $\pi(x, y)$ that is still in $C(v_2)$, let $x_{p(3)}$ be the vertex succeeding it on $\pi(x, y)$, and let v_3 be the nearest 3-vertex to $x_{p(3)}$. And so on all the way to $x_{p(c-1)}$ and v_{c-1}

Recall that $d(x_{p(1)}, v_1) \leq p(1)$. It is not hard to show by induction that we always have

$$d(x_{p(i)}, v_i) \leq p(i)$$

We have already proved the base case. Now say that it is true for i. We know that $d(x_{p(i+1)}, v_{i+1}) \leq d(x_{p(i+1)}, v_i)$ because otherwise we would have $x_{p(i+1)} \in C(v_i)$, which would contradict how we chose $x_{p(i+1)}$. But by the inductive hypothesis,

$$d(x_{p(i+1)}, v_i) \leq d(x_{p(i+1)}, x_{p(i)}) + d(x_{p(i)}, v_i) \leq$$
$$(p(i+1) - p(i)) + p(i) = p(i+1)$$

Now, note that we can always take the following path to any $x_{p(i)}$ (see Figure 2):

$$(x, x_{p'(1)}) \circ (x_{p'(1)}, x_{p(1)}) \circ (x_{p(1)}, v_1) \circ (v_1, x_{p'(2)}) \circ$$
$$(x_{p'(2)}, x_{p(2)}) \circ (x_{p(2)}, v_2) \circ ... \circ (x_{p'(i)}, x_{p(i)})$$

It is easy to see that this path is no longer than the path that follows $\pi(x, y)$ but takes detours from $x_{p(1)}$ to v_1 and back to $x_{p(1)}$, from $x_{p(2)}$ to v_2 and back, and so on until $x_{p(i-1)}$ to v_{i-1}. Thus, we have a path to $x_{p(i)}$ of length at most

$$p(i) + 2[d(x_{p(1)}, v_1) + ... + d(x_{p(i-1)}, v_{i-1})]$$

But recall that $d(x_{p(i)}, v_i) \leq p(i)$, so our path to $x_{p(i)}$ has length at most

$$p(i) + 2[p(1) + ... + p(i-1)]$$

Thus, intuitively, we want to know when

$$p(i) + 2[p(1) + ... + p(i-1)] \leq (1+\epsilon)p(i)$$

because then we have a good path to $x_{p(i)}$.

Definition 7.2. Define $\beta(i) = (3/\epsilon)^i \alpha$. Note that $\beta(0) = \alpha$ (see Definition 7.1 for α)

It is not hard to check that the following holds: [2]

$$\beta(i) > (2/\epsilon)(\beta(0) + \beta(1) + ... + \beta(i-1))$$

so if we let j be the *first* index $(1 \leq j \leq c-1)$ for which $p(j) \geq \beta(j-1)$ then the above path to $p(j)$ is a $(1+\epsilon)$ path. Moreover, we have $p(j) \geq \beta(j-1) \geq \alpha$, so $p(j)$ is far away enough from x to satisfy our needs (recall that we wanted to get past x_α to ensure that we went at least a $1/d$ fraction of the way to y). Thus, we are set if such an index j exists.

If such an index does not exist then we have $p(j) < \beta(j-1)$ for all $1 \leq j \leq c-1$. But note that v_{c-1} is a $(c-1)$-vertex so its cluster contains every

[2]this stems from the identity that for any $C > 1$, we have that $C^1 + C^2 + ... + C^{i-1} = (C^i - 1)/(C-1) < C^i/(C-1)$

vertex; in particular, the edge (v_{c-1}, y) is in H. This means that we can take the following $x-y$ path in H:

$$(x, x_{p'(1)}) \circ (x_{p'(1)}, x_{p(1)}) \circ (x_{p(1)}, v_1) \circ (v_1, x_{p'(2)}) \circ (x_{p'(2)}, x_{p(2)}) \circ (x_{p(2)}, v_2) \circ ... \circ (v_{c-1}, y)$$

The length of this path is at most

$$\begin{aligned} & d(x,y) + (2p(1) + 2p(2) + ... + 2p(c-1)) \\ & \leq d(x,y) + 2[\beta(0) + \beta(1) + ... + \beta(c-2)] \\ & \leq d(x,y) + \epsilon\beta(c-1) \end{aligned}$$

Thus, we simply need to ensure that $d(x,y) + \epsilon\beta(c-1) \leq (1+\epsilon)d(x,y)$, or equivalently, that $\beta(c-1) \leq d(x,y)$. But recall our initial assumption that $\pi(x,y)$ has at least $d = (3/\epsilon)^c$ edges, and hence has length at least d (we assumed that all edges have length at least 1 – see Section 2). This implies that

$$\beta(c-1) = (3/\epsilon)^{c-1}\alpha < d\alpha = d\lfloor d(x,y)/d \rfloor \leq d(x,y)$$

This completes the proof since we have shown that with only $O(c)$ edges we can find a $(1+\epsilon)$ path that either gets us at least a $1/d$ fraction of the way to y, or takes us directly to y.

Truncated Clusters: We are almost done, except that we now need to prove that truncating our clusters does not affect the proof. Since we only need to prove a bound on AUD_k, we assume that $d(x,y) \leq 4k$. We now notice that at no point in the proof did we ever use an edge of weight more than $2d(x,y)$. Thus, if we truncate our clusters at $\epsilon k/2$ then the proof above works as long as $d(x,y) \leq \epsilon k/4$.

This suggests that we should split $\pi(x,y)$ into paths of length $\epsilon k/4$. We let y_1 be the furthest vertex on $\pi(x,y)$ that is at distance at most $\epsilon k/4$ from x. We let y_2 be the furthest vertex at distance at most $\epsilon k/4$ from y_1, and so on up to some y_r. Note that $r = O(4k/(\epsilon k/4)) = O(1/\epsilon)$, so we are splitting $\pi(x,y)$ into a constant number of paths of length $\leq \epsilon k/4$. (technical note: it might not be possible to thus split $\pi(x,y)$ in weighted graphs – take, for example, a single heavy edge. We omit the details but we can trivially overcome this by labeling a few especially heavy edges as their own subpath of $\pi(x,y)$.)

Each of these subpaths is short enough that using truncated clusters is identical to using regular ones. Thus, to get from x to y we use the same method as with non-truncated clusters to get from x to y_1, then from y_1 to y_2, and so on until we get to y (see Figure 3). Each of the individual paths is a $(1+\epsilon)$ path, so the resulting $x-y$ path is too. Note that we might use $O((3/\epsilon)^c \log n)$ edges on *each* subpath from y_i to y_{i+1}, so this technique multiplies our bound on the AUD_k of H by a constant ($O(1/\epsilon)$) factor.

■

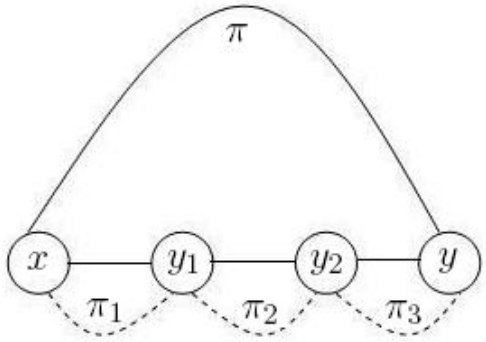

Figure 3. This figure shows how truncating clusters changes the paths we take in H. π is the $x-y$ path that we would get if we used non-truncated clusters. π may not exist in H because we use truncation, but $\pi_1 \circ \pi_2 \circ \pi_3$ is guaranteed to exist.

8. Conclusion

We have presented a fully dynamic algorithm for $(2+\epsilon)$ approximate all pairs shortest paths that achieves a close to linear update time and a $O(\log\log\log n)$ query time. It would be nice if one could polish this result: reduce the query time to constant, reduce the update time to $\widetilde{O}(m)$, remove the super-polynomial dependence on ϵ, and if possible, remove the dependence on the ratio between edge weights. An important related problem is to develop an efficient fully dynamic algorithm for $(1+\epsilon)$ shortest paths.

References

[1] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," *J. ACM*, vol. 51, no. 6, pp. 968–992, 2004.

[2] S. Even and Y. Shiloach, "An on-line edge deletion problem," *J. ACM*, vol. 28, no. 1, pp. 1–4, 1981.

[3] V. King, "Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs," in *Proc. of the 40th FOCS*, 1999, pp. 81–91.

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

[5] M. Thorup, "Fully-dynamic all-pairs shortest paths: faster and allowing negative cycles," in *Proc. of the 9th SWAT*, 2004, pp. 384–396.

[6] M. Thorup and U. Zwick, "Approximate distance oracles," *J. ACM*, vol. 52, no. 1, pp. 1–24, 2005.

[7] ——, "Spanners and emulators with sublinear distance errors," in *Proc. of the 17th SODA*, Miami, Florida, 2006, pp. 802–809.