Near Linear Time $(1+\epsilon)$-Approximation for Restricted Shortest Paths in Undirected Graphs

Aaron Bernstein*

September 29, 2011

Abstract

We present a significantly faster algorithm for the *restricted* shortest path problem, in which we are given two vertices s, t, and the goal is to find the shortest path that is subject to a side constraint. More formally, rather than just having a single weight, each edge has two weights: a cost-weight, and a delay-weight. We are given a threshold T which corresponds to the maximum delay we can afford, and the goal is to find the $s-t$ path that minimizes total cost while still having delay-length at most T.

There are many applications for this problem, as it can model situations where we need a path that has to achieve some balanced trade off between two different parameters. The exact version of the problem is known to be NP-hard [3], but there has been a series of results on $(1+\epsilon)$ approximations, which culminated in a $\widetilde{O}(mn)$ algorithm for general graphs in 1999 [4, 8]. We present the first result to break though this barrier, achieving a close to linear running time – technically it is $\widetilde{O}(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$. It does have several drawbacks, however. It is randomized (Las Vegas), it only works for undirected graphs, and it approximates *both* parameters (previous algorithms found a $(1+\epsilon)$ shortest path with delay *exactly* T or less, or a *shortest* path with delay at most $(1+\epsilon)T$, whereas our algorithm incurs a $(1+\epsilon)$ approximation on both counts.) Our result presents an entirely new approach to the problem, which could potentially be generalized to work for directed graphs and to approximate only one of the parameters.

1 Introduction

A classical problem in graph algorithms is to find the shortest path between two vertices s, t. But in some situations, not all shortest paths are feasible, as there are other side constraints to take into account. The *restricted* shortest path problem (also known as the *bicriteria* or *constrained* shortest path problem) models this situation by introducing edges that have not just one weight (*e.g.* cost), but *two* weights: a cost-weight and and a delay-weight. Given a delay threshold T, the goal is to find an $s-t$ path that minimizes the cost-length while having a delay-length of at most T (the cost/delay length of a path is the sum of the cost/delay weights of its edges).

This problem is applicable to scenarios in which we need a path to achieve some trade-off between two parameters. In particular, it is often used in QoS (quality of service) routing, where the goal is to route a package along the cheapest possible path while also satisfying some quality constraint for the user. For example, we might impose a limit on the total delay of the path (*i.e* how long it takes), or on the amount of packet loss. (See *e.g.* [7])

1.1 Previous Results There exist many practical algorithms for this problem and its variations (see [5, 13] for a small sample), but theoretically speaking the restricted shortest path problem (RSP) is known to be NP-hard in the exact case [3]. There does, however, exist a series of approximation results [4, 6, 8, 12] for this problem which culminated in two state of the art algorithms. The first, by Lorenz and Raz [8], runs in $\widetilde{O}(mn/\epsilon)$ time [1] and returns an $s-t$ path with delay-length at most T (the threshold is preserved exactly), and cost-length at most $(1+\epsilon)$ larger than the optimum. Goel *et al.* [4] present a result that runs in $O(mn/\epsilon)$ time and returns a path with cost-length no longer than the optimum, but with delay-length up to $(1+\epsilon)T$. It also has the added advantage that in $O(mn)$ time it computes the shortest restricted paths from s to *every* possible destination t.

1.2 Our Contributions Our algorithm is the first to go beyond this barrier, achieving a close to linear update time. Technically, it is $O(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$ which is $o(mn^{\delta})$ for any fixed δ (assuming constant ϵ). Our algorithm does have several drawbacks, however: it is randomized (Las Vegas), it only works for *undirected graphs*, and it incurs a $(1+\epsilon)$ approximation in both parameters. That is, it returns a path with delay length at most $(1+\epsilon)T$ and cost length within $(1+\epsilon)$ of the shortest path with threshold T.

Like the result of Goel *et al.*, our algorithm computes approximate shortest distances (with threshold $(1+\epsilon)T$) from source s to *all* destinations $t \in V$, and can return any particular $s-t$ path in

*Columbia University, New York, NY, 10027. email: bernstei@gmail.com. Supported by the NSF GRFP Fellowship.

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

$O(L)$ time, where L is the number of edges on the path. We achieve our improved running time by presenting an entirely new approach to the restricted shortest path problem (see Section 3). It is quite possible that this approach could be further extended to work for directed graphs, or to only incur a $(1+\epsilon)$ approximation in one of the parameters.

1.3 Related Work The natural generalization of the restricted shortest path (RSP) problem is known as the multi-constrained shortest path problem, where instead of having a single side constraint (the delay threshold T), we have many different side constraints, each with their own corresponding weight parameter and threshold. There is a large literature on this problem; see *e.g.* [9, 14].

Another generalization: in classical RSP, the user fixes a specific delay threshold T, and we compute the optimum path satisfying that threshold. But what if there are many users, each with their own desired cost-delay trade off, and hence their own threshold? In this case, rather than looking for a single path, we are looking for a small collection of paths that approximately represents all the different trade offs one could achieve. This is known as computing the *approximate Pareto curve*. There have been several results on this problem, with the state of the art being a recent algorithm by Diakonikolas and Yannakakis [2]. Their algorithm requires a classical RSP algorithm (which takes a fixed threshold T) as a subroutine, so our improved algorithm yields faster running times for constructing the Pareto curve in undirected graphs.

2 Notation

We start with an *undirected* graph $G = (V, E)$ with m edges and n vertices. Each edge on the graph has two *non-negative* weights: a cost-weight and a delay-weight. We let $w_c(u, v)$ denote the cost-weight of an edge (u, v), and $w_d(u, v)$ denote the delay weight. Given a path P, we let $c(P)$ denote the cost-length of this path (the sum of the edge cost-weights), and $d(P)$ denote the delay-length.

For any vertices x, y, we define the shortest T-path between x and y to be the path that has minimum cost-length among all $x - y$ paths with delay-length at most T, and we define the shortest T-distance between x and y to be cost-length of this path. We let $P^T_{x,y}$ denote this shortest T-path, $c^T(x, y)$ denote its cost-length, and $d^T(x, y)$ denote its delay-length. Note that $d^T(x, y)$ is at most T, but it may be smaller.

Given some $x - y$ path P, we say that an $x - y$ path P^* is a α, β approximation to P if $c(P^*) \leq \alpha c(P)$ and $d(P^*) \leq \beta d(P)$. We say that $P^*_{x,y}$ is a α, β-shortest T-path if $c(P^*_{x,y}) \leq \alpha c^T(x, y)$ and $d(P^*_{x,y}) \leq \beta T$

In the restricted shortest paths (RSP) problem, we are given a source s and a delay-threshold T, and our goal is to compute, for every destination t, the cost and delay length of some $(1+\epsilon), (1+\epsilon)$-shortest T-path from s to t. We can also return the actual path, but for simplicity, we only focus on distances. (Note that we will sometimes end up with approximations like $(1+\epsilon)^2$, or $(1+4\epsilon)$, but these are equivalent to $(1+\epsilon)$,as we can just use a smaller ϵ – *e.g* $\epsilon' = \epsilon/4$.)

Much of our algorithm will avoid looking directly at the original graph G, and will instead work with an *emulator* H, which is a graph on the same vertices (but with different edges) that has a similar distance structure to G but is easier to work with. The use of emulators is extremely common in approximate graph algorithms, but unlike the vast majority of papers, we do not use them in order to sparsify the original graph, but rather to create shortest paths with only a small number of edges.

Definition 2.1. *Given some threshold T', we say that a graph H has a T'-approximate hop diameter of h, denoted T'-AHD, if given any two vertices x, y and any threshold $T^* < T'$ there exists a $(1+\epsilon), (1+\epsilon)$-shortest T^*-path from x to y that has at most h edges (the weight of these edges is not relevant). When the threshold T' we are working with is clear in context, as is usually the case, we simply write AHD instead of T'-AHD*

Note that in the above definition, we are requiring there to be few-edge approximations for all pairs x, y and all possible thresholds $T^* < T'$, not just T'-paths. We only consider thresholds $T^* < T'$ because when working with a threshold T', we do not care about paths of larger threshold.

3 General Approach

In this section, we outline our general approach, and highlight the new steps we take in order to go beyond the $\widetilde{O}(mn)$ barrier. The structure of all RSP algorithms, including ours, relies on the following "naive" approach

Naive Algorithm: Let s be our source, let T be our threshold, and let $D = \max_{t \in V}\{c^T(s, t)\}$ – *i.e*, D is the largest relevant cost-length. *Assuming integer weights*, we can compute *exact* shortest T-distances from s to every destination t in min{O(mD), O(mT)} time. For details, see Section 2 of [6].

Thus, we already have an algorithm that works well for small weights. In fact, it is instructive to consider what we call *half-unweighted* graphs, where the delay-weights are still arbitrary, but cost-weights are all 1 (or vice versa). On such graphs, $D \leq n$, so the naive algorithm runs in $O(mn)$ time. In general graphs with large weights, however, D and T might be very large.

All previous algorithms used rounding and scaling to adapt the naive $O(mD)/O(mT)$ algorithm to work for large weights. There was quite a bit of work to do – introducing a way to scale the naive algorithm, picking the right rounding/scaling factor, *etc.* – but everything they did fit into this same basic framework. On the plus side, this meant that their algorithms were clean and technically simple. But this also introduced a fundamental limitation: the best one can hope for by scaling large weights is reducing the general case to the half-unweighted case, and in fact, on a half-unweighted graph all these algorithms essentially reduce to the naive one. But even in this case the naive algorithm takes $O(mn)$ time, so if we want to go beyond this, we need to introduce an entirely new approach.

At its core, our result consists of a faster approach to the half-unweighted case (the hard part), which we have generalized to arbitrary weights through rounding and scaling (the easy part).

3.1 The Two Main Theorems

On a conceptual level, our algorithm uses the framework we developed in an earlier paper to solve the seemingly unrelated problem of dynamic shortest paths [1]. This framework allows us to transform an efficient algorithm for *very* small distances (the O(mT) algorithm) into an algorithm for general graphs. In particular, we do the following:

1. Find an algorithm that works well when distances are extremely small ($O(mT)$ is indeed efficient when T is very small. Note that it is not enough for all delay-weights to be 1, as this would yield $T = n$. We need not just delay-weights but delay-*distances* to be small.)

2. Construct an emulator H that models distances in the original graph G, but has small T-approximate hop diameter, where T is the original threshold (see Definition 2.1).

3. Use a simple weight-scaling technique to turn the graph H with small hop lengths into a graph H' with small *weighted* distances (not just small edge weights, but small distances).

4. Run the algorithm for small distances on H'

We formalize this approach with the following two theorems, which we prove later.

Theorem 3.1. *(step 3) Say that we are given a graph G, a source s, and a threshold T, and let h be the T-AHD of G. Then, in $O(m)$ time, we can create a new graph G' with the following properties.*
1. *The edges of G' are identical to those of G, except with different delay-weights. For one, the delay-weights in G' are natural numbers.*
2. *Given any two vertices x, y, and Setting $T' = (1+2\epsilon)h/\epsilon \leq 3h/\epsilon = O(h)$, the shortest T'-path from x to y in G' is a $((1+\epsilon),(1+\epsilon))$-approximate shortest T-path from x to y in G.*

Theorem 3.2. *(step 2) Given a graph G, we can create an emulator H with exactly the same restricted distances, but with AHD $\widetilde{O}((\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$. The time to construct H is $\widetilde{O}(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$, and the number of edges is $\widetilde{O}(m + n(\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$*

Corollary 3.1. *(steps 1, 4) Given a graph G, a source s, and a threshold T, we can compute $((1+\epsilon),(1+\epsilon))$ T-shortest paths from s in time $\widetilde{O}(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$*

Proof. (of corollary) First we construct an emulator H with small AHD using Theorem 3.2, and then modify its edge weights to yield a new graph H' with the properties described in Theorem 3.1. But note that because H has small AHD, we have $T' = 2h/\epsilon = \widetilde{O}((\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$, which means we can use the naive $O(mT')$ algorithm to compute T'-shortest paths from s in H' in time $\widetilde{O}(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$. By property 2 of Theorem 3.1, these paths in H' are $((1+\epsilon),(1+\epsilon))$-shortest T-paths in H, which has the same distances as G, so these paths are in fact $((1+\epsilon),(1+\epsilon))$ approximations of the shortest T-paths from s in G.

Note that on its own, Theorem 3.1 yields a $O(mh) = O(mn)$ algorithm, where h is the T-AHD of the original graph G; first we create a graph G' with $T' = (1+2\epsilon)h/\epsilon = O(h)$, and then we run the $O(mT')$ algorithm. This is exactly the result of [4], and in fact the proof we use is conceptually similar, except that instead of incorporating the scaling into the naive $O(mT)$ algorithm, we scale the graph itself. The scaling we do is straight-forward; all the difficulties come from Theorem 3.2 (step 2). It is this theorem that allows us to overcome the $O(mn)$ barrier, because instead of just scaling large weights, we change the underlying structure of the graph by reducing the AHD.

Remark: The framework presented above (reducing AHD, then scaling) comes from our earlier

FOCS 2009 result [1] on dynamic shortest paths. But this is only a conceptual framework, and the actual implementation differs enormously depending on the specific problem at hand.

In our FOCS 2009 result, the main difficulties lay in making sure that things work out in a dynamic setting, where everything is constantly changing. However, we could afford to be sloppy with how we actually built the emulator. In both the current paper and in our earlier FOCS 2009 paper, the algorithm has two time-consuming steps: constructing the emulator (step 2 above), and actually computing the desired paths in this emulator (step 4 above). In our FOCS 2009 paper, the bottleneck was step 4, so we had plenty of time for the emulator construction. But in this problem step 2 is the bottleneck, so not only do we have less time to construct the emulator, but our emulator has to preserve not just shortest, but *restricted* shortest paths. Thus, to keep the algorithm efficient, we need to construct a more complex emulator in less time, which forces us develop a more inventive construction than in the FOCS 2009 paper.

4 Proof of Theorem 3.1

We now prove Theorem 3.1 from the previous section. All we do is round and scale the delay-weights of the edges.

Proof. Recall that T is the threshold, and h is the AHD (see Definition 2.1) of the graph G we are currently working with (possibly not the original graph). We want to somehow scale down all the delay weights while ensuring that the resulting weights are integral. To do this:

1. Round all edge delay-weights up to the nearest integer multiple of $\epsilon T/h$, and denote the resulting graph G^*. That is, if an edge had cost-weight x, delay-weight y in G, then we round up y (but not x) to get the corresponding weight in G^*. For G^* we will use the threshold $T^* = (1+2\epsilon)T$, which is where the approximation error arises.

2. Divide all edge delay-weights in G^* by $\epsilon T/h$, and denote the resulting graph G'. Note that all edge weights in G' are integral. Define T' to be $(1+2\epsilon)T/(\epsilon T/h) = (1+2\epsilon)h/\epsilon = O(h)$ – to get the new threshold T' we just scale the old threshold T^* down by our scaling factor.

(Note that this rounding and scaling always results in a graph G' with natural weights, regardless of whether the original graph had natural weights.)

The second step only involves scaling down by a multiplicative factor, and so does not change the structure of the graph at all. Thus, it is clear that a T'-shortest path in G' is a T^*-shortest path in G^* (T' is just $T*$ scaled down). Creating G^* from G, however, introduces an additive factor of up to $\epsilon T/h$ on each edge, which *does* change the structure. We need to show that this only introduces a $(1+2\epsilon)$ approximation. More formally, let P be the shortest T-path from x to y in G, and let P^* be the shortest T^*-path from x to y in G^* ($T^* = (1+2\epsilon)T$). Note that since G and $G*$ have the same edges (with different weights), $P*$ can also be viewed as a path in G. To complete the proof, we need to show P^* is a $((1+\epsilon),(1+2\epsilon))$ approximation to P – that is, we should have $c(P^*) \leq (1+\epsilon)c(P)$, and $d(P^*) \leq (1+2\epsilon)T$.

By definition, since P^* is a T^*-path we have $d(P^*) \leq T^* = (1+2\epsilon)T$. To prove that $c(P^*) \leq (1+\epsilon)c(P)$, recall that since G has AHD h, there must be some $((1+\epsilon),(1+\epsilon))$ shortest T-path Q from x to y in G, such that Q has at most h edges. That is, $c(Q) \leq (1+\epsilon)c(P)$ and $d(Q) \leq (1+\epsilon)T$. Now, let Q^* be the path Q when viewed in G^* – same edges, but with slightly different weights. The cost-weights are the same in G and G^* so we still have $c(Q^*) = c(Q) \leq (1+\epsilon)c(P)$. As for delays, the delay-weight of each edge on Q went by at most $\epsilon T/h$, so since Q has at most h edges, the total additive error is at most ϵT. Thus, $d(Q^*) \leq d(Q) + \epsilon T \leq (1+\epsilon)d(P) + \epsilon T \leq (1+2\epsilon)T = T^*$. Thus, we know there exists a T^*-path in G^* with cost length at most $(1+\epsilon)c(P)$, so P^*, being the *shortest* T^*-path, will also have cost at most $(1+\epsilon)c(P)$.

Thus, we have shown that at the cost of a $(1+\epsilon),(1+2\epsilon)$ approximation we can work in G^* instead of G. We can make this a $(1+\epsilon),(1+\epsilon)$ approximation by just using $\epsilon' = \epsilon/2$ instead of ϵ. The whole point of this is that G^* is much easier to work with because we can scale it down to G', which has a smaller threshold $T' = (1+2\epsilon)h/\epsilon \leq 3h/\epsilon = O(h)$ (see step 2 above). If h is small, then this will allow the naive $O(mT') = O(mh)$ algorithm to run very quickly on G'.

5 Reducing The Hop Diameter.

We now turn to proving Theorem 3.2, in which we must construct an emulator H of G that has small AHD. The basic idea is that since hop diameter refers only the number of edges, without regard to their weight, we can reduce it by adding "shortcut" edges to the graph. For example, if a path from x to y uses $1,000$ edges and has cost-length c, delay-length d, then we can shortcut this path by adding a single edge from x to y of cost-weight c, delay-weight d.

There are two main obstacles to this approach. The first is that it is unclear which paths we should even try to shortcut. We cannot simply shortcut the "shortest" paths, because we have a trade-off between cost and delay. Do we shortcut the path from x to y with high cost but low delay, or the one with high delay but low cost? Note that we know our overall delay from s should be around T, but this does not tell us anything about what delay we should aim for on some $x-y$ subpath. Our solution will be to shortcut many different paths from x to y in order to capture all of the trade offs we might want.

The much more significant obstacle is that in order to shortcut the path from x to y, we need to compute the restricted shortest path from x to y – even if we knew to aim for delay d, we would need to find the shortest d-path. But this whole paper is about how to compute the restricted shortest path, so we seem to have a circular algorithm. We overcome this by observing that although computing restricted shortest paths is hard in general, it is easy when the path from x to y has small delay – we can just use the O(mT) algorithm. Our approach is thus to first compute some of the easy paths, then use this information to create shortcut edges, which in turn reduce the hop diameter and allow us to compute slightly less easy paths, which yield even more shortcuts edges, and so on. We never get to the point of computing *all pairs* restricted shortest paths, but we do end up computing enough shortcut edges to drastically reduce the AHD.

Our general outline (Section 3) suggested that our algorithm is broken up into two discrete parts: first we reduce the AHD, then we scale the weights down. But in fact, the two steps are intermingled, where first we scale down, then reduce the AHD a bit, then scale down a bit more, then reduce the AHD, and so on. We now formalize what we need from a single such iteration.

Definition 5.1. *Let* $\alpha = (\frac{2}{\epsilon})^{O(\sqrt{log(n)})}$.

Lemma 5.1. *Say that we are given a graph* $G_i = (V, E_i)$ *(likely not the original graph) with threshold* T_i *and positive integer delay-weights. We can construct an emulator* H_i *on the same vertex set with the following properties*

1. H_i *contains all the edges of* G_i *plus some shortcut edges, and* all *restricted distances in* H_i *are identical to those of* G_i *(shortcut edges do not change the distance structure).*

2. H_i *has* T_i*-AHD* $h_i \leq \max\{\alpha, \epsilon T_i/6\}$ *(see Definition 2.1 for* T_i*-AHD)*

3. H_i *has* $|E_i| + \widetilde{O}(n\alpha)$ *edges*

4. H_i *can be constructed in* $\widetilde{O}(|E_i|\alpha)$ *time.*

We leave the proof for later, and first observe the following:

Corollary 5.1. *Assuming this lemma we can construct the small AHD emulator H of Theorem 3.2*

Proof. (of corollary) All graphs have hop-diameter $\leq n$, so our first step is to use Theorem 3.1 to scale G down to a new graph G_1 with threshold $T_1 = O(n)$. By Theorem 3.1, we can now focus on computing shortest T_1-paths in G_1, as these will be $((1+\epsilon),(1+\epsilon))$ shortest T-paths in G. Note that by Theorem 3.1 the delay-weights in G_1 are natural numbers, and they remain so throughout our algorithm.

We now use Lemma 5.1 to construct a new graph H_1 with the four properties listed above. In particular, H_1 has AHD $h_1 \leq \max\{\alpha, \epsilon T_1/6\}$: if we ever get down to an AHD of α then we are done (our main goal is to obtain a small AHD), so we assume that $h_1 \leq \epsilon T_1/6$. But this means that we can use Theorem 3.1 to scale H_1 down to a new graph G_2 with threshold $T_2 \leq 3h_1/\epsilon \leq (3\epsilon T_1/6)/\epsilon = T_1/2$. We can now focus on computing shortest T_2-paths in G_2 because they will be $(1+\epsilon),(1+\epsilon)$ shortest T_1 paths in H_1, which be know by property 1 of Lemma 5.1 has the same distance structure as G_1.

Thus, we have reduced the problem of working in graph G_1 with threshold T_1, to that of working in G_2 with threshold $T_2 \leq T_1/2$. We continue in this fashion, cutting the threshold in half at every step. The next step is to use Lemma 5.1 to create a graph H_2 with AHD $h_2 \leq \epsilon T_2/6$, and then use Theorem 3.1 to create a graph G_3 with threshold $T_3 = 3h_2/\epsilon \leq T_2/2$.

We keep reducing the threshold and AHD in this fashion, until withing $O(\log(n))$ iterations we reach a graph H_k with $h_k \leq \alpha$. H_k serves as the desired small AHD emulator H in Theorem 3.2. All we have left is to analyze the construction time, number of edges, and approximation error required by this theorem. By property 3 of Lemma 5.1 each iteration adds $\widetilde{O}(n\alpha)$ edges, so after $O(\log(n))$ such iterations, the resulting graph H_k has $\widetilde{O}(m + n\alpha)$ edges, as desired. By property 4 of lemma 5.1, the construction time in each iteration is $\widetilde{O}(|E_i|\alpha)$, but since the number of edges is always $\widetilde{O}(m+n\alpha)$, the time of an iteration never exceeds $\widetilde{O}(m\alpha + n\alpha^2) = \widetilde{O}(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)})})$, yielding a total time of $\widetilde{O}(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)})})$ over all iterations.

As for approximation, we incur a $(1+\epsilon)$ error every time we use Theorem 3.1, so since we have $\log(n)$ iterations the total error is $(1+\epsilon)^{\log(n)}$. To reduce this, we use $\epsilon' = \epsilon/4\log(n)$, which yields $(1+\epsilon')^{\log(n)} = (1+\epsilon/4\log(n))^{log(n)} \leq (1+\epsilon)$. *This is* *the only place where we had to reduce ϵ by more than a constant, and this results in the extra* $\log\log(n)$ *factor in the exponent of our overall running time.*

We have shown how to use Lemma 5.1 to construct the small emulator H of Theorem 3.2 with AHD $h = O((\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$. Recall that once we have this the problem is easy because we can just use Theorem 3.1 one last time to get a graph H' with threshold $T' = O(h) = O((\frac{2}{\epsilon})^{O(\sqrt{log(n)}\log\log(n))})$ and then use the naive $O(mT')$ algorithm. We now turn to the final step: proving Lemma 5.1.

6 Proof of Lemma 5.1

6.1 Clustering

Our proof for Lemma 5.1 is partly based on sampling and clustering techniques of Thorup and Zwick [10], as well as on the emulator they construct with these techniques [11]. But although this serves as our foundation, we end up using these techniques for a seemingly unrelated problem, and we introduce many new techniques and ideas. Firstly, we have to generalize everything to work for *restricted* shortest paths, which introduces several new difficulties as these are much harder to work with. Secondly, we use these techniques to a very different end: where the Thorup and Zwick paper focused on constructing *sparse* emulators, our focus is on reducing the hop diameter, which requires an entirely different analysis.

The basic idea of Thorup and Zwick is to let different vertices have different *priorities*, where high priority vertices are rarer but also better connected.

Definition 6.1. *Let $V = A_0 \supseteq A_1 \supseteq ... \supseteq A_{r-1} \supseteq A_r = \emptyset$ be sets of vertices ($r = O(log(n))$ 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 $c^B(v, A_i)$ to be the shortest B-distance from v to some i-vertex: $c^B(v, A_i) = \min_{w\in A_i/A_{i+1}} c^B(v, w)$. (See Section 2 for $c^B(v, w)$). We define the i^B-witness of v to be corresponding i-vertex: the i^B witness of v is $\text{argmin}_{w\in A_i/A_{i+1}} c^B(v, w)$ (if there is a tie between several w, pick an arbitrary one to be the i^B witness).*

Definition 6.2. *We define the B-cluster of an i-vertex v, denoted $C^B(v)$, to contain all vertices w for which the B-distance from w to v is smaller than the B-distance from w to its nearest $i + 1$ vertex: $C^B(v) = \{w \mid c^B(w, v) < c^B(w, A_{i+1})\}$. We define the edges of this cluster to be all edges with at least one endpoint in $C^B(v)$. We define $|C^B(v)|$ to be the number of vertices in this cluster, and $E[C^B(v)]$ to be the number of edges.*

We use the same sets A_i as Thorup and Zwick [10]: we start with $A_0 = V$, and every vertex in A_i is independently sampled and put into A_{i+1} with probability $1/n^{1/r}$. This leads to:

Lemma 6.1. *Given any vertex w, and any threshold B, we have that with high probability w is only contained in $\widetilde{O}(rn^{1/r}) = \widetilde{O}(n^{1/r})$ B-clusters ($r = O(log(n))$ is subsumed into the $\widetilde{O}$ notation).*

Proof. Let us first fix some priority i, and look at the number of clusters $C^B(v)$, *for i-vertices v*, that contain w. To do this, let $v_1, v_2, ..., v_k$ be the list of vertices in A_i (*i.e* priority i or higher) sorted by increasing B-distance from w. Because of how we sampled our vertices, each of these vertices has a $1/n^{1/r}$ chance of being in A_{i+1}. Thus, by a simple application of the Chernoff bound, we have that with high probability, one of the first $\widetilde{O}(n^{1/r})$ vertices in this list is in A_{i+1}. But by definition of clusters, this means that w is not in the cluster $C^B(v_j)$ for any v_j that is further from w – *i.e* comes after this vertex from A_{i+1} in our list. Thus, with high probability w is in the cluster of only $\widetilde{O}(n^{1/r})$ i-vertices. There are r different priorities, so since the above holds for each of them, we have that in total w is in $\widetilde{O}(rn^{1/r}) = \widetilde{O}(n^{1/r})$ clusters (with high probability).

Remark 6.1. *This claim is true with high probability for any vertex w and any threshold B, so by the union bound it is true with high probability* for all *vertices w and thresholds $B < n$. Thus, we will assume for the rest of the paper that the above lemma always holds.*

Corollary 6.1. *For any particular B, the total size of all B-clusters is $\widetilde{O}(n^{1+1/r})$, and the total number of edges in all B-clusters is $\widetilde{O}(mn^{1/r})$. Thus, the overall number of vertices, totaled among all B-clusters for $B = 1, 2, ..., \beta$ is $\widetilde{O}(\beta n^{1+1/r})$, and the overall number of edges is $\widetilde{O}(\beta mn^{1/r})$ (β is an upper bound of our choosing).*

Proof. Each vertex is in $\widetilde{O}(n^{1/r})$ B-clusters, so the total size of all B-clusters is clearly n times this. Similarly, the total number of edges is $\widetilde{O}(n^{1/r}\Sigma_{v\in V}\text{degree(v)}) = \widetilde{O}(mn^{1/r})$.

6.2 Using The Clusters

Definition 6.3. *Define*

$$\beta = \sqrt{\log(n)} \cdot (7/\epsilon)^{\sqrt{\log(n)}+4}$$

Let $r = \sqrt{\log(n)}$ and note that β and $n^{1/r}$ are both $\widetilde{O}((\frac{2}{\epsilon})^{O(\sqrt{log(n)})})$

We show in the next section how to compute the clusters, but first we show how to use them to prove

Lemma 5.1. Recall that we start with a graph G_i with threshold T_i and want to construct an emulator H_i that has AHD $h_i \leq \textsc{max}\{(\frac{2}{\epsilon})^{O(\sqrt{log(n)})}, \epsilon T_i/6\}$, in addition to a few other properties.

H_i consists of all of the edges of G_i, plus some new "shortcut" edges. In particular, for every vertex v and every threshold B with $1 \leq B \leq \beta$, we add an edge from v to every vertex w in $C^B(v)$. The weight of this edge is precisely the weight of the shortest B-path from v to w. That is, if $P^B(v, w)$ has length $(c^B(v, w), d^B(v, w))$, then we set edge (v, w) to have weight $(c^B(v, w), d^B(v, w))$. Similarly, for every vertex v, every priority i, and every threshold $B < \beta$ we add a shortcut edge from v to the i^B witness of v (see Definition 6.1 for i^B-witness).

Our shortcut edges clearly never decrease distances, so property 1 of Lemma 5.1 holds. The total number of shortcut edges added is just the total size of all the clusters used, which by the second corollary of Lemma 6.1 is $\widetilde{O}(\beta n^{1+1/r}) = \widetilde{O}(n(\frac{2}{\epsilon})^{O(\sqrt{log(n)})})$ (the total number of i^B-witnesses is even smaller – only $O(r\beta n)$). We must now show that the clusters can be computed efficiently, and that these shortcut edges sufficiently reduce the AHD.

6.3 Computing The Clusters The clusters are defined in terms of $c^B(v, A_i)$ (see Definition 6.1), so first we compute that.

Lemma 6.2. *Given a threshold B and a priority i, we can compute $c^B(v, A_i)$, for all vertices v, in a total of $O(mB)$ time.*

Proof. We create a slightly modified graph G' by adding a dummy source s', and including an edge of cost-weight 0 and delay-weight 0 from s' to every vertex in A_i. To get from s' to v in G' we must go through one of the vertices in A_i, so the shortest path from s' to v will go through the vertex in A_i that is closest to v. Thus, all we do is compute shortest B-paths from s': for every vertex v, the cost-length of this path is precisely $c^B(v, A_i)$, and the first vertex after s' is the i^B-witness of v. We can compute these B-paths from s' in $O(mB)$ time using the naive algorithm.

Corollary 6.2. *For all priorities i and all thresholds B with $1 \leq B \leq \beta$ we can compute all of the $c^B(v, A_i)$ in a total of $\widetilde{O}(r\beta^2 m) = \widetilde{O}(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)})})$ time (see Definition 6.3 for r and β): we just sum the $O(mB)$ running time for all possible priorities i and thresholds B.*

Now, to compute the clusters $C^B(v)$ we rely on the following lemma that allows us to build high-threshold clusters from lower-threshold ones.

Lemma 6.3. *Let v be some i-vertex, and let w be any vertex in cluster $C^B(v)$. Let P be the shortest B-path from from v to w, and δ be the delay-length of this path (we know $\delta \leq B$). Let w' be the vertex right before w on P, let P' be the subpath of P from v to w', and let δ' be the delay-length of this path. Lemma: We must have that w' is in $C^{\delta'}(v)$.*

Proof. Recall from Section 2 that $w_c(w', w)$ denotes the cost-weight of edge (w, w'), $w_d(w', w)$ denotes the delay-weight, and $c(P)$ denotes the cost-length of path P. Since (w', w) is the only difference between paths P' and P we must have $w_c(w', w) = c(P) - c(P')$ and $w_d(w', w) = \delta - \delta'$

Now, Say for contradiction that w' is not in $C^{\delta'}(v)$. Then, by definition of clusters, there must be some $i + 1$-vertex v' such that $c^{\delta'}(v', w') \leq c^{\delta'}(v, w')$. But then consider the $v' - w$ path that follows the shortest δ'-path from v' to w', followed by edge (w', w) This path has cost-length $c^{\delta'}(v', w') + w_c(w', w) \leq c^{\delta'}(v, w') + w_c(w', w) = c(P) = c^B(v, w)$ and has delay-length $\leq \delta' + w_d(w', w) = \delta \leq B$. Thus, it is a B-path from v' – an $i + 1$-vertex – to w with cost-length $\leq c^B(v, w)$, so w is *not* in $C^B(v)$, which is a contradiction.

Definition 6.4. *We say that we "compute distances" within a cluster $C^B(v)$ if we compute $c^B(v, w)$ for every vertex $w \in C^B(v)$.*

Lemma 6.4. *Given any vertex v and an upper threshold β, we can compute clusters $C^1(v), C^2(v), ..., C^\beta(v)$, and the distances within them in a total of $\widetilde{O}(\beta \sum_{B=1}^{\beta} E[C^B(v)])$ time.*

Proof. Say that v is an i-vertex. The basic idea is to increment the threshold by one at each step: first we compute distances in $C^1(v)$, then use this to compute distances in $C^2(v)$, then $C^3(v)$, all the way up to $C^\beta(v)$.

Recall the notations $w_c(v, w)$ and $w_d(v, w)$ from Section 2. Computing $C^1(v)$ is easy because it can only contain v and its neighbors. We just look at all edges leaving v, and we consider vertices w for which $w_d(v, w) = 1$ (to ensure we do not break our delay threshold). We then add w to $C^1(v)$ if $w_c(v, w) < c^1(w, A_{i+1})$, which we computed above: v is an i-vertex, so this directly checks the cluster property. Computing distances within the cluster is also trivial as the cost-distance to a vertex w in $C_1(v)$ is just $w_c(v, w)$

Now, say that we have already computed clusters $C^1(v), ..., C^k(v)$ and distances within them. We show how to use this to compute $C^{k+1}(v)$. For every j with $1 \leq j \leq k$ we consider all edges (u, w) where u is in $C^j(v)$ (it is fine if w is also in $C^j(v)$, though it does not have to be). For every such edge (u, w) we include w in $C^{k+1}(v)$ if properties 1 and 2 below hold.

1. $w_d(u, w) \leq k + 1 - j$.

2. $c^j(v, u) + w_c(u, w) < c^{k+1}(w, A_{i+1})$ (i is the priority of v).

What we do: If these properties hold, we include w in $C^{k+1}(v)$. We compute distances by setting $c^{k+1}(v, w) = c^j(v, u) + w_c(u, w)$. *However, if we also end up including w in C^{k+1} through some different edge (u', w) with u' in some $C^{j'}(v)$, then we naturally take the minimum of the distances found.*

The first requirement ensures that the $v - w$ path going through edge (u, w) does not break our delay-threshold of $k + 1$: the path from v to u has delay at most j (u is in $C^j(v)$), and edge (u, w) adds at most $k + 1 - j$ more delay. The second ensures that w is closer to v than to the nearest $i + 1$-vertex (the cluster requirement). Thus, any vertex added by our algorithm does in fact belong in $C^{k+1}(v)$.

We now show that given any w in $C^{k+1}(v)$, our algorithm correctly puts w in the cluster, and finds the shortest $k + 1$ distance from v to w. Let P be the shortest $k + 1$-path from v to w, let w' be the vertex before w on P, and let P' be the subpath of P from v to w'. Let δ be the delay-length of P and let δ' be the delay length of P'. We clearly have $\delta' < \delta \leq k + 1$. By lemma 6.3 w' is in $C^{\delta'}(v)$, so our algorithm will look at (w', w) when it looks at $C^{\delta'}(v)$; it is possible that it will also look at edge (w', w) in other clusters, *e.g.* if w' is in $C^{\delta'-1}(v)$, but we want to focus on $C^{\delta'}(v)$ in particular. We know that $w_d(w', w) = \delta - \delta' \leq k + 1 - \delta'$ so requirement 1 holds. Requirement 2 will also hold because $c^{\delta'}(v, w') + w_c(w', w)$ is precisely the cost-length of P, which is the *shortest* $k + 1$-path from v to w. Thus, since we are assuming that w is in $C^{k+1}(v)$, by definition of clusters we have that the $c^{\delta'}(v, w') + w_c(w, w') = c^{k+1}(v, w) < c^{k+1}(w, A_{i+1})$. Similarly, when we set $c^{k+1}(v, w) = c^{\delta'}(v, w') + w_c(w', w)$ (see "What we do" above) this gives us the cost-length of P, which is in fact the shortest $k + 1$-distance from v to w

Computing any particular $C^{k+1}(v)$ requires us to look at every edge in $C^1(v), C^2(v), ..., C^k(v)$, yielding a running time of $O(\sum_{B=1}^{k} E[C^B(v)]) \leq O(\sum_{B=1}^{\beta} E[C^B(v)])$ (recall that $\beta = \sqrt{\log(n)} \cdot (7/\epsilon)^{\sqrt{\log(n)}+4}$ is the upper bound on the cluster thresholds we look at). Thus, the time to compute all of $C^1(v), ..., C^\beta(v)$ is just $\widetilde{O}(\beta \sum_{B=1}^{\beta} E[C^B(v)])$, as desired.

COROLLARY 6.3. *The* total *time to compute clusters $C^B(v)$ and distances withing these clusters, for all $v \in V$ and all B such that* $1 \leq B \leq \beta$ *is* $\widetilde{O}(\beta^2 mn^{1/r}) = \widetilde{O}(m(\frac{2}{\epsilon})^{O(\sqrt{log(n)})})$.

Proof. The above lemma tells us the running time for a single vertex. For all vertices it is

$$\widetilde{O}(\beta \sum_{v \in V} \sum_{B=1}^{\beta} E[C^B(v)]) = \widetilde{O}(\beta \sum_{B=1}^{\beta} \sum_{v \in V} E[C^B(v)])$$

$$\leq_{\text{by Lemma 6.1}} \widetilde{O}(\beta \sum_{B=1}^{\beta} mn^{1/r}) = \widetilde{O}(\beta^2 mn^{1/r})$$

6.4 Analyzing the approximate hop diameter In Lemma 5.1 – the lemma we are proving – we start with some graph G' that has threshold T'. We constructed the desired emulator H' by keeping all the edges of G' and also adding shortcut edges from every vertex v to every vertex $w \in C^B(v)$, and every i^B-witness of v, for all threshold $B \leq \sqrt{\log(n)} \cdot (7/\epsilon)^{\sqrt{\log(n)}+4}$ – see Section 6.2 for details. We have already verified that H' satisfies properties 1,3, and 4 of Lemma 5.1, so all we have left is to prove the following: *The emulator H' described above has a T'-approximate hop diameter (T'-AHD) of* $\max\{\sqrt{\log(n)} \cdot (7/\epsilon)^{\sqrt{\log(n)}+4}, \epsilon T'/6\}$ (see Definition 2.1 for T'-AHD).

For the rest of this section, we let $r = \sqrt{\log(n)}$ be the number of vertex priorities (see Definition 6.1), let $\beta = \sqrt{\log(n)} \cdot (7/\epsilon)^{\sqrt{\log(n)}+4} = (\frac{2}{\epsilon})^{O(\sqrt{log(n)})}$ be the upper limit on our cluster thresholds, and let $h' = \text{MAX}\{\beta, \epsilon T'/6\}$ be the AHD that we are shooting for. Note that $h' = \beta$ signifies that the AHD is small enough to terminate (see Corollary of Lemma 5.1), so the real case to consider is $h' = \epsilon T'/6$ (we show later exactly where the β comes in).

Let us focus on some arbitrary pair of vertices x, y, and let P' be the shortest T'-path between them in G'. Note that since H' contains all the edges of G', P' is also the shortest T'-path in H', but it might have more than h' edges. We want to prove that H' contains a $((1 + \epsilon), (1 + \epsilon))$ approximation to P' – call it P^* – with at most h' edges. *Note that we do not need an algorithm for actually finding this path: We are only trying to prove that such a path P^* exists.* Our proof presents a constructive method for finding this path, but it is not something the computer ever runs

Our AHD analysis borrows from our earlier FOCS 2009 paper [1], but a lot of extra work is required to extend it to restricted shortest paths. The proof is quite involved, so we start with some intuition, and then present the details.

Intuition: The idea of course is that P^* will use some of the shortcut edges in H to go a long way with just a few edges. We are only scaling the AHD down by $6/\epsilon$, so the shortcut edges do not even have to shortcut all that much. We start with x, and look at clusters $C^B(x)$ for various different B. If one of these clusters contains many vertices on P', then we simply take a shortcut edge from x to some x' far down P': this guarantees a lot of progress with a single edge.

But there is no guarantee that a cluster goes far down P'. If there is some higher priority vertex w near P' then by definition of clusters, $C^B(x)$ will *not* go far down P'. But if there is a higher priority vertex w nearby, we can get to it with just a few shortcut edges. This might involve taking a detour, as w is likely not on P' itself, but as long as w is near the path, the detour will be small. We now continue from w, and the point is that since it has higher priority, clusters $C^B(w)$ will be larger, and hence allow us to make more progress towards y.
So in short, we keep on taking steps towards y by using our cluster shortcut edges. At each step we have two cases. Either the cluster is large, in which case we can make a lot of progress, or it is small, in which case there is a high priority vertex nearby, so we can get to it without incurring much of an error. There are only r priorities, so we will not spend more than $O(r)$ edges climbing in priority; if we ever get to priority $r-1$ (the highest priority), we will be guaranteed to make a whole lot of progress from there.

Details: As mentioned in the intuition, each shortcut edge will either climb a priority, or make a lot of progress down P'. Note that making a lot of progress is important for two reasons. Firstly, it ensures that we are using a small number of edges to go a long way, thus keeping the AHD small. But secondly, it makes up for the earlier detours that we made: the detours incurred some small error, but this error can be subsumed into a multiplicative $(1+\epsilon)$ approximation if we later make a lot of progress.

The issue is that it is unclear how to define progress because we have two weight-parameters: does progress mean a shortcut edge with large delay weight, or large cost weight? If we take an edge with large delay weight, this will make up for previous detours in terms of delay, but it might not make up for them in terms of cost. To overcome this, we note that there are three sorts of paths: ones that have large cost relative to their delay (cost-heavy), ones with large delay relative to their cost (delay-heavy), and ones that are approximately balanced. The idea is that when working with a cost-heavy path, we care much more about getting a good approximation to the cost: we don't mind a bad delay approximation, as the overall delay of the path is relatively small, so even a 3-approximation to the delay would not lead to much delay error. Similarly, when working with a delay-heavy path, we focus on getting a good delay approximation. When working with a balanced path, a lot of cost-progress is also a lot of delay-progress, so we need a good approximation on both counts.

As in the intuition, say that we are trying to get from x to y, and let P' be the shortest T'-path between them. Recall that $d(P')$ is the delay of this path, and that $c(P')$ is its cost. We are trying to find a path P^* that approximates P' but has $\leq \epsilon T'/6$ edges.

DEFINITION 6.5. *For any very u on P', let $P'_{x,u}$ be the subpath of P' from x to u. Let $c(u)$ be the cost of this path, and let $d(u)$ be the delay of this path.*

DEFINITION 6.6. *Let $\rho = c(P')/d(P')$ (ρ for ratio). This will serve as a scaling factor between cost and delay.*

What we want: Starting from x, at each step we want to take a path from x to some u that makes a lot of progress with a small number of edges, and is a good approximation to $P'_{x,u}$. We will then restart the process from u,taking a good path to some u_2 that takes us even closer to y. continuing in this way, we will get a good path from x to y.

More formally, we want to find a path $P^*_{x,u}$, from x to some u on P', such that $P^*_{x,u}$ satisfies <u>one</u> of the following three cases.

Case 1 (cost-heavy):

1. $P^*_{x,u}$ contains $\leq 3r$ edges. ($P^*_{x,u}$ does not contain too many edges)
2. $c(u) \geq \rho(36r/\epsilon)$ (going to u makes enough cost progress towards y)
3. $c(u) \geq \rho(3/\epsilon)d(u)$ (the subpath $P'_{x,u}$ is cost-heavy)
4. $c(P^*_{x,u}) \leq (1+\epsilon)c(P'_{x,u})$ ($P^*_{x,u}$ is a good cost-approximation)
5. $d(P^*_{x,u}) \leq 3d(P'_{x,u})$ ($P^*_{x,u}$ is an OK delay-approximation)

Case 2 (delay-heavy):

1. $P^*_{x,u}$ contains $\leq 3r$ edges. ($P^*_{x,u}$ does not contain too many edges)

2. $d(u) \geq (36r/\epsilon)$ (going to u makes enough delay progress towards y)

3. $d(u) \geq (3/\epsilon)c(u)/\rho$ (the subpath $P'_{x,u}$ is delay-heavy)

4. $d(P^*_{x,u}) \leq (1+\epsilon)d(P'_{x,u})$ ($P^*_{x,u}$ is a good delay-approximation)

5. $c(P^*_{x,u}) \leq 3c(P'_{x,u})$ ($P^*_{x,u}$ is an OK cost-approximation)

Case 3 (balanced approximation):

1. $P^*_{x,u}$ contains $\leq 3r$ edges. ($P^*_{x,u}$ does not contain too many edges)

2. $c(u) \geq \rho(36r/\epsilon)$ (going to u makes enough cost progress towards y)

3. $d(u) \geq (36r/\epsilon)$ (going to u makes enough delay progress towards y)

4. $c(P^*_{x,u}) \leq (1+\epsilon)c(P'_{x,u})$ ($P^*_{x,u}$ is a good cost-approximation)

5. $d(P^*_{x,u}) \leq (1+\epsilon)d(P'_{x,u})$ ($P^*_{x,u}$ is also a good delay-approximation)

Why this is enough: Say that we had a method such that starting from x, we were guaranteed to find a vertex u on P', and a path $P^*_{x,u}$ satisfying one of the three cases above. We then restart the process from u, and find a path P^*_{u,u_2} to some u_2 further down P'. We would continue in this fashion until reaching y. We now want to show that this would path be a $(1+\epsilon),(1+\epsilon)$ approximation to P' (the shortest T'-path from x to y), that has at most $\epsilon T'/6$ edges.

To see that the path contains at most $\epsilon T'/6$ edges, note that since the total cost of P' is $c(P') = \rho d(P') \leq \rho T'$, there can be at most $c(P')/(\rho(36r/\epsilon)) \leq \epsilon T'/(36r)$ subpaths of cost-length $\rho(36r/\epsilon)$. Thus, we can take a total of at most $\epsilon T'/(36r)$ paths of cases 1 and 3. By an analogous argument for delay, there can be a total of at most $\epsilon T'/(36r)$ subpaths of cases 2 and 3. Thus, there are at most $\epsilon T'/(18r)$ subpaths in total, and by property 1 of all the cases, each such subpath has at most $3r$ edges, yielding at most $\epsilon T'/6$ edges in total.

We now show that the path is a $(1+\epsilon),(1+\epsilon)$ approximation. Let us first focus on the cost approximation. We know that the paths of case 1 and 3 yield $(1+\epsilon)$ cost approximations, so the total cost-error from paths of case 1 and 3 is at most $\epsilon c(P')$. But what about paths of case 2? Recall that all these paths are approximations of various $P'_{x,u}$ – subpaths of P'. The total delay of all these subpaths of P' (the ones we approximate with case 2) is clearly at most $d(P')$, but since all the subpaths are delay-heavy (property 3 of case 2), the total *cost* of all these subpaths is at most $d(P')/((3/\epsilon)/\rho) = c(P')/(3/\epsilon)$. Thus, since our case 2 paths all yield a 3-approximation to the cost (case 2, property 5), the total cost error from case 2 paths is $\leq 3c(P')/(3/\epsilon) = \epsilon c(P')$. Thus, the total error from all these cases is at most $2\epsilon c(P')$, which yields a $(1+2\epsilon)$ cost-approximation. Using $\epsilon' = \epsilon/2$ yields the desired $(1+\epsilon)$ cost approximation.

The proof for a $(1+\epsilon)$ delay approximation is exactly symmetrical, so we omit it. The problem case for delay is course case 1 instead of case 2.

How to find a suitable path $P^*_{x,u}$ All we have left is to show that there always exists a path $P^*_{x,u}$ that satisfies one of the three cases above. We will prove its existence by constructing it.

Let w'_1 be the furthest vertex on P' for which w'_1 is in cluster $C^{d(w'_1)}(x)$, and let w_1 be the vertex right after w'_1 on P' (see Definition 6.5 for $d(w'_1)$). Note that since w_1 is not in $C^{d(w_1)}(x)$, there must be some 1-vertex v_1 such that $c^{d(w_1)}(w_1, v_1) \leq c^{d(w_1)}(w_1, x)$ (in particular, let v_1 be the $1^{d(w_i)}$ witness of w_1 – see Definition 6.1). Now, let w'_2 be the furthest vertex on P' that is in $C^{d(w'_2)}(v_1)$, and let w_2 be the vertex right after w'_2 on P'. Let v_2 be the $2^{d(w_2)}$-witness of w_2; so we have $c^{d(w_2)}(w_2, v_2) \leq c^{d(w_2)}(w_2, v_1)$. Let w'_3 be the furthest vertex on P' that is in $C^{d(w'_3)}(v_2)$, and define w_3, and 3-vertex v_3 analogously. Continue in this fashion until we get up to w'_{r-1}, w_{r-1}, and $(r-1)$-vertex v_{r-1} ($r-1$ is the largest possible priority). Because of how we picked w_1, we have $c^{d(w_1)}(w_1, v_1) \leq c^{d(w_1)}(w_1, x) = c(w_1)$ (see Definition 6.5 for $c(w_1)$). We can prove by induction that for all i we have

$$c^{d(w_i)}(w_i, v_i) \leq c(w_i)$$

We have already proved the base case of $i = 1$. Now, we assume it is true for some i, and prove it for $i+1$. We know that $c^{d(w_{i+1})}(w_{i+1}, v_{i+1}) \leq c^{d(w_{i+1})}(w_{i+1}, v_i)$ because otherwise w_{i+1} would be in $C^{d(w_{i+1})}(v_i)$, which contradicts how we picked w_{i+1}. Also, note that we can get from v_i to w_{i+1} by going down to w_i and then following P' from w_i to w_{i+1} ; the subpath of P' from w_i to w_{i+1} has cost $c(w_{i+1}) - c(w_i)$. Thus, we must have that

$$\begin{aligned} & c^{d(w_{i+1})}(w_{i+1}, v_{i+1}) \leq c^{d(w_{i+1})}(w_{i+1}, v_i) \\ & \leq c(w_{i+1}) - c(w_i) + c^{d(w_i)}(w_i, v_i) \\ & \leq_{\text{by induction}} c(w_{i+1}) - c(w_i) + c(w_i) \\ & = c(w_{i+1}) \end{aligned} \tag{6.1}$$

(Technical note: we must also show that $d(w_{i+1})$ is enough delay to get from v_i to w_{i+1}, but this can be shown via an identical induction argument, so we omit it.) Now, note that we can always take the following shortcut edges to any w_i (see Figure 1)

$$(x, w'_1) \circ (w'_1, w_1) \circ (w_1, v_1) \circ (v_1, w'_2) \\ \circ (w'_2, w_2) \circ (w_2, v_2) \circ ... \circ (w'_i, w_i)$$

All of these shortcut edges exist in our emulator: for every edge (v_{i-1}, w'_i) we have that w'_i is in the cluster $C^{d(w'_i)}(v_{i-1})$; every edge (w'_i, w_i) is in the original graph; and for every edge (w_i, v_i) we have that v_i is a $i^{d(w_i)}$-witness of w_i.

It is easy to see that this path is no longer than the path that follows P' from x to w_i, but takes detours from w_1 to v_1 and back to w_1, from w_2 to v_2 and back, and so on, the last detour being from w_{i-1} to v_{i-1} and back (note that a detour from some v_j to w_j is up to twice the path-length from v_j to w_j because it goes from v_j to w_j and then back). Thus, we have a path from x to w_i – which we call the *standard path to* w_i – of cost length at most

$$\begin{aligned} & c(w_i) + 2[c^{d(w_1)}(w_1, v_1) + c^{d(w_2)}(w_2, v_2) + \\ & ... + c^{d(w_{i-1})}(w_{i-1}, v_{i-1})] \\ & \leq_{\text{by Equation 6.1}} \\ & c(w_i) + 2[c(w_1) + c(w_2) + ... + c(w_{i-1})] \end{aligned} \tag{6.2}$$

Similarly, the delay of this path is at most

$$d(w_i) + 2[d(w_1) + d(w_2) + ... + d(w_{i-1})]$$

Thus, it is easy to see that this path is always a 3-approximation to both cost and delay. In order to get a good cost approximation we need $c(w_i) >> 2[c(w_1)+c(w_2)+...+c(w_{i-1})]$, and similarly for a good delay approximation we need $d(w_i) >> 2[d(w_1) + d(w_2) + ... + d(w_{i-1})]$.

Definition 6.7. *Define*

$$B(i) = (7/\epsilon^2)^i \cdot (36r/\epsilon)(3/\epsilon)$$

$$B_\rho(i) = \rho(7/\epsilon^2)^i \cdot (36r/\epsilon)(3/\epsilon) = \rho B(i)$$

Note that $B(0) = (36r/\epsilon)(3/\epsilon), B_\rho(0) = \rho(36r/\epsilon)(3/\epsilon)$, *and that since the number of priorities (r) is at most* $\sqrt{\log(n)}$ *we have* $B(r-1) \leq B(\sqrt{\log(n)} - 1) < \sqrt{\log(n)} \cdot (7/\epsilon)^{\sqrt{\log(n)}+4}$, *which is precisely* β *– our upper bound on cluster thresholds.*

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

$$B(i) \geq (6/\epsilon^2)(B(0) + B(1) + ... + B(i-1))$$

$$B_\rho(i) \geq (6/\epsilon^2)(B_\rho(0) + B_\rho(1) + ... + B_\rho(i-1))$$

Now, let j be the *first* index for which *at least one* of the following properties holds (assume for now that such an index j exists).

1. $c(w_j) \geq B_\rho(j-1)$
2. $d(w_j) \geq B(j-1)$

Key: Note that since j is the first such index, for every index j' before j we have $c(w_{j'}) < B_\rho(j'-1)$ and $d(w_{j'}) < B(j'-1)$. We have four cases to consider.

Case 1 (cost-heavy): $c(w_j) \geq B_\rho(j-1)$ **and** $c(w_j) \geq \rho(3/\epsilon)d(w_j)$
We claim that in this case the path fits into Case 1 from the "what we want" section above. We now check each of the five properties. Property 1 holds because our standard path to w_j uses 3 shortcut edges to climb from priority i to $i+1$ $((v_i, w'_{i+1}) \circ (w'_{i+1}, w_i) \circ (w_i, v_{i+1}))$, so since there are r priorities, it uses at most $3r$ edges. Property 2 holds because $c(w_j) \geq B_\rho(j-1) \geq B_\rho(0) = \rho(36r/\epsilon)(3/\epsilon) > \rho(36r/\epsilon)$. Property 3 holds because of our assumption for this case. Property 4 holds because the path we take to w_j has cost $c(w_j) + 2[c(w_1) + ... + c(w_{j-1})] \leq c(w_j)+2[B_\rho(0)+B_\rho(1)+...B_\rho(j-2)] \leq c(w_j)+\epsilon c(w_j)$, where the last inequality is true because $c(w_j) \geq B_\rho(j-1) \geq (6/\epsilon^2)(B_\rho(0) + B_\rho(1) + ... + B_\rho(j-2))$. Property 5 holds because as mentioned before, our standard path to any w_i is automatically at least a 3-approximation in both cost and delay (at the very worst, the detours are twice the path length).

Case 2 (delay-heavy): $d(w_j) \geq B(j-1)$ **and** $d(w_j) \geq (3/\epsilon)c(w_j)/\rho$
In this case, the path fits into Case 2 from the "what we want" section. The proof is exactly analogous to the proof for Case 1 above, so we omit it.

Case 3 (balanced): ***either*** $c(w_j) \geq B_\rho(j-1)$ **or** $d(w_j) \geq B(j-1)$ **(or both)** **<u>and</u>** $c(w_j)/(3/\epsilon) \leq \rho d(w_j) \leq c(w_j)(3/\epsilon)$
In this case, we show that the path fits into Case 3 from the "what we want" section above. Let us say, without loss of generality, that $c(w_j) \geq B_\rho(j-1)$ (the case where $d(w_j) \geq B(j-1)$ is symmetrical). We know that our standard path to w_i contains $\leq 3r$ edges, so Property 1 holds. Property 2 holds because $c(w_j) \geq B_\rho(j-1) \geq B_\rho(0) = \rho(36r/\epsilon)(3/\epsilon) > \rho(36r/\epsilon)$. Property 3 holds because we know from the balance assumption for this case that

$$d(w_j) \geq c(w_j)/(\rho(3/\epsilon)) \geq B_\rho(0)/(\rho(3/\epsilon)) = (36r/\epsilon)$$

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

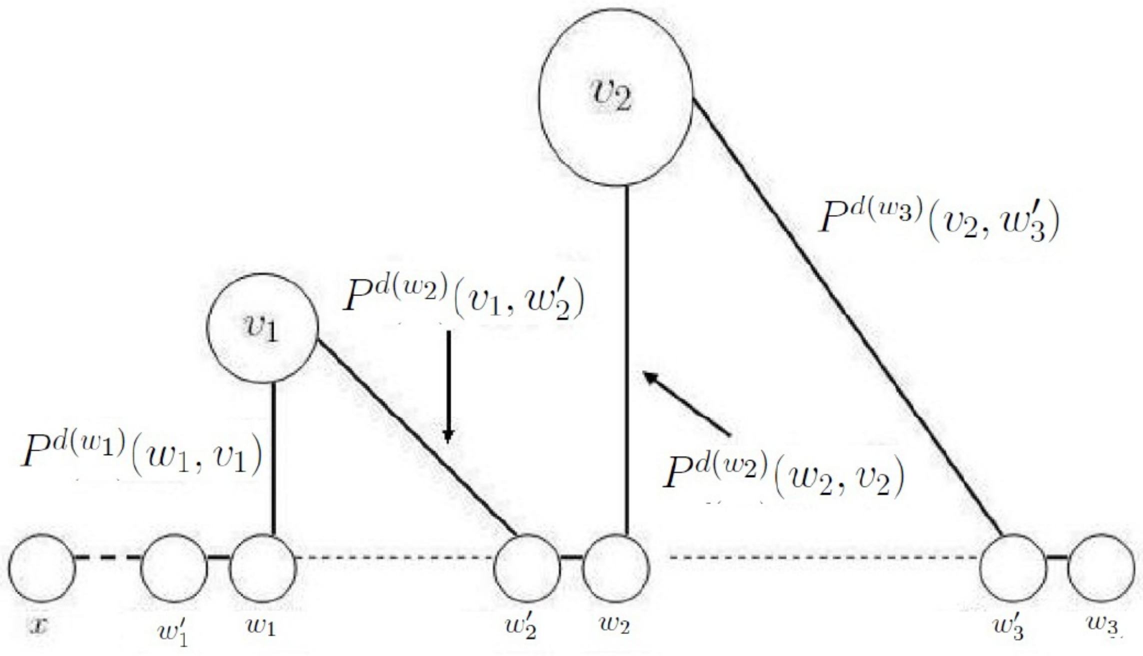

Figure 1: This figure shows our standard path in H from x to w_i that contains few edges. The dotted path is P', the actual shortest path from x to w_i , while the bold path is the one that we use. Each bold line corresponds to a single shortcut edge.

Property 4 holds because $c(w_j) \geq B_\rho(j-1)$ so the path we take to w_j has cost $c(w_j)+2[c(w_1)+c(w_2)+ ...+c(w_{j-1})] \leq c(w_j)+2[B_\rho(0)+B_\rho(1)+...B_\rho(j-2)] \leq c(w_j)+\epsilon c(w_j)$. Finally, property 5 holds because

(6.3)
$$\begin{aligned} d(w_j) &\geq c(w_j)/(\rho(3/\epsilon)) \geq B_\rho(j-1)/(\rho(3/\epsilon)) \\ &= B(j-1)/(3/\epsilon) \\ &\geq (6/\epsilon^2)(B(0)+B(1)+...+B(j-2))/(3/\epsilon) \\ &\geq (2/\epsilon)(B(0)+B(1)+...+B(j-2)) \end{aligned}$$

Thus, the path we take to w_j has delay

$$\begin{aligned} &d(w_j)+2[d(w_1)+...+d(w_{j-1})] \\ &\leq d(w_j)+2[B(0)+B(1)+...+B(j-2)] \\ &\leq d(w_j)+\epsilon d(w_j) \end{aligned}$$

Why index j exists: The three cases above assumed the existence of an index j with $c(w_j) \geq B_\rho(j-1)$ or $d(w_j) \geq B(j-1)$. We now prove such an index exists. Say, for contradiction, that we got to v_{r-1} without encountering such a j. Since there are no r-vertices ($r-1$ is the highest priority), we have that for any threshold B, cluster $C^B(v_{r-1})$ includes all vertices within delay distance B of v_{r-1}. In particular, the vertex w_r will be at least β delay-distance away from v_{r-1} (recall: β is our upper bound on cluster thresholds), because otherwise it would be in $C^{d(w_r)}(v_{r-1})$, contradicting how we chose w_r. Hence, if we get that far without finding a suitable index j, then we will necessarily have $d(w_r) \geq \beta > B(r-1)$, so $j = r$ will be our desired index.

We have shown that our standard path to w_i always fits one of the three cases from the "what we want" section, so by our proof in the "why this is enough" section, this guarantees a $(1+\epsilon),(1+\epsilon)$ shortest T'-path from x to y with at most $\epsilon T'/6$ edges.

There are two last technical caveats to cover. Firstly, as our path to w_i progressed, the shortcut edges got bigger and bigger. Could this not pose a problem, since we only computed clusters up to threshold $\beta = \sqrt{\log(n)} \cdot (7/\epsilon)^{\sqrt{\log(n)}+4}$? It does not pose a problem because the largest delay distance we ever needed to cover for the above cases to work out was $d(w_j) = B(r-1) < \beta$. Thus, if the delay from v_{j-1} to w'_j – our usual shortcut edge – is ever bigger than β, the we can safely go from v_{j-1} to some closer w^*_j that is β units away, as this will still satisfy all the properties of the three cases.

Another caveat is that our argument relies on the notion that it is okay for us to take some detours when we climb in vertex priority because we can make up for them later. But this requires that the original path P' is long enough for us to have the space to do this. In particular, the largest distance we ever needed to cover for the approximation analysis for work out was $B(r-1) < \beta$, so we need the original path P' from x to y to have delay distance at least β. But if it has delay-distance less than β then it must have less than β edges (Recall that all delay-weights we work with are natural numbers because the first step of our alorithm is to apply the transformation of Theorem 3.1 to the original graph), so we can just finish off the last small segment of P' by directly following the path P' (no shortcut edges), hence using only $\beta = (\frac{2}{\epsilon})^{O(\sqrt{log(n)})}$ additional edges. This is why in Lemma 5.1, we can only guarantee an AHD of $\textsc{max}\{\epsilon T'/6, (\frac{2}{\epsilon})^{O(\sqrt{log(n)})}\}$.

7 Concluding Remarks

We have presented a new approach to the restricted shortest path problem in undirected graphs that allowed us to break through a long-standing $O(mn)$ barrier and achieve a near-linear running time. See section 3 for an overview of our techniques. This paper is only a first step, however, and there are many problems left to be solved. We hope our framework could provide a direction for some of them.

1. Is there an $o(mn)$ algorithm for *directed* graphs? To use our approach, one would need to construct a small hop diameter emulator for directed graphs. Note that whereas sparsification is provably impossible in directed graphs – even with approximation – this is not the case for reducing the hop diameter. We already know several emulators that do so, but we only know how to construct them in $O(mn)$ time, not $o(mn)$ time.

2. Is there an $o(mn)$ algorithm that approximates only the cost, while *exactly* preserving the threshold? To use our approach, one would need to construct an emulator that reduces not just the approximate hop diameter, but the *exact* hop diameter. Such emulators exist, but as in the directed case, it takes us $O(mn)$ time to construct them.

3. Can we polish up our algorithm to remove the $(\frac{2}{\epsilon})^{O(\sqrt{log(n)} \log\log(n))}$ and achieve a running time of $\widetilde{O}(mn/\text{poly}(1/\epsilon))$?

4. Can the ideas in our algorithm be applied to the *multi-constrained* shortest path problem? (This is the natural generalization of the restricted shortest path problem – see Section 1.3).

8 Acknowledgments

I would like to thank Professor Mihalis Yannakakis for some very fruitful discussions, and my friend Raju Krishnamoorthy for his thoughtful suggestions.

References

[1] A. Bernstein. Fully dynamic approximate all-pairs shortest paths with query and close to linear update time. In *Proc. of the 50th FOCS*, pages 50–60, Atlanta, GA, 2009.

[2] I. Diakonikolas and M. Yannakakis. Small approximate pareto sets for biobjective shortest paths and other problems. *SIAM J. of Computing*, pages 1340–1371, 2009.

[3] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP Completeness.* W.H.Freeman, San Francisco, 1979.

[4] A. Goel, K. Ramakrishnan, D. Kataria, and D. Logothetis. Efficient computation of delay-sensitive routes from one source to all destinations. *INFOCOOM 2001*, 2.

[5] G. Handler and I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309, 1980.

[6] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17:36–42, 1992.

[7] D. H. Lorenz and A. Orda. Qos routing in networks with uncertain parameters. *IEEE/ACM Transactions on Networking*, 6:768–778, 1998.

[8] D. H. Lorenz and D. Raz. A simple efficient approximation scheme for the restricted shortest path problem. *Operations Research Letters*, 28:213–219, 1999.

[9] G. Rosario and T. Luca. A survey on multi-constrained optimal path computation: Exact and approximate algorithms. *Comput. Netw.*, 54:3081–3107, December 2010.

[10] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[11] M. Thorup and U. Zwick. Spanners and emulators with sublinear distance errors. In *Proc. of the 17th SODA*, pages 802–809, Miami, Florida, 2006.

[12] A. Warburton. Approximation of pareto optima in multiple-objective shortest path problems. *Operations Research*, 35:70–79, 1987.

[13] G. Xue and S. K. Makki. Multi-constrained qos routing: a norm approach. *IEEE Transactions on Computers*, 56(6):859–863, 2007.

[14] G. Xue, W. Zhang, J. Tang, and K. Thulasiraman. Polynomial time approximation algorithms for multi-constrained qos routing. *IEEE/ACM Trans. Netw.*, 16:656–669, June 2008.