A Nearly Optimal Algorithm for Approximating Replacement Paths and k Shortest Simple Paths in General Graphs

Aaron Bernstein *

Abstract

Let $G = (V, E)$ be a directed graph with positive edge weights, let s, t be two specified vertices in this graph, and let $\pi(s,t)$ be the shortest path between them. In the *replacement paths* problem we want to compute, for every edge e on $\pi(s,t)$, the shortest path from s to t that avoids e. The naive solution to this problem would be to remove each edge e, one at a time, and compute the shortest $s-t$ path each time; this yields a running time of $O(mn + n^2 \log n)$. Gotthilf and Lewenstein [8] recently improved this to $O(mn+n^2 \log\log n)$, but no $o(mn)$ algorithms are known.

We present the first *approximation* algorithm for replacement paths in directed graphs with positive edge weights. Given any $\epsilon \in [0, 1)$, our algorithm returns $(1 + \epsilon)$-approximate replacement paths in $O(\epsilon^{-1} \log^2 n \log(nC/c)(m+n \log n)) = \widetilde{O}(m \log(nC/c)/\epsilon)$ time, where C is the largest edge weight in the graph and c is the smallest weight.

We also present an even faster $(1 + \epsilon)$ approximate algorithm for the simpler problem of approximating the k shortest simple $s-t$ paths in a directed graph with positive edge weights. That is, our algorithm outputs k different simple $s-t$ paths, where the kth path we output is a $(1 + \epsilon)$ approximation to the actual kth shortest simple $s-t$ path. The running time of our algorithm is $O(k\epsilon^{-1} \log^2 n(m + n \log n)) = \widetilde{O}(km/\epsilon)$. The fastest *exact* algorithm for this problem has a running time of $O(k(mn+n^2 \log\log n)) = \widetilde{O}(kmn)$ [8]. The previous best approximation algorithm was developed by Roditty [15]; it has a stretch of 3/2 and a running time of $\widetilde{O}(km\sqrt{n})$ (it does not work for replacement paths).

Note that all of our running times are nearly optimal except for the $O(\log(nC/c))$ factor in the replacements paths algorithm. Also, our algorithm can solve the variant of approximate replacement paths where we avoid vertices instead of edges.

1 Introduction

1.1 The Problem A fundamental problem in graph algorithms is to find the shortest path between two given points s and t. A natural generalization is to consider the problem where edges occasionally fail. More formally, let $\pi(s,t)$ be the shortest path from s to t. In the *replacement paths* problem, we want to compute, for every edge e to $\pi(s,t)$, the shortest $s-t$ path that avoids e.

*Massachusetts Institute of Technology; Cambrdige, MA, 02139. Email: bernstei@gmail.com

Although there exist several efficient algorithms for computing replacement paths in certain classes of graphs, no fast algorithms are known for weighted, directed graphs. The naive approach would be to remove each edge e on $\pi(s,t)$, one at a time, and compute the shortest $s-t$ path each time; this takes $O(mn+n^2 \log n)$ time. Gotthilf and Lewenstein [8] recently improved this to $O(mn + n^2 \log\log n)$, but no $o(mn)$ algorithms are known.

One of the main motivations for replacement paths is that the problem of computing the k shortest simple $s-t$ paths in a graph can be reduced to running k executions of a replacement paths algorithm. Thus, any $O(T(n))$ algorithm for replacement paths yields a $O(kT(n))$ algorithm for the k shortest simple paths. As far as we know, all the fastest algorithms for computing the exact k shortest simple paths rely on this reduction.

Since no $o(mn)$ algorithm is known for replacement paths in general graphs, we improve efficiency by settling for an *approximation*. In particular, for every edge e on $\pi(s,t)$, we want to return an α approximation to the shortest $s-t$ path that avoids e, where α is our stretch factor. It is easy to see that any α-approximate, $O(T(n))$ algorithm for computing replacement paths also yields a $O(T(n))$ algorithm for computing an α-approximation to the second shortest simple path (just take the shortest replacement path). Moreover, Roditty showed in [15] that this in turn yields an α-approximate, $O(kT(n))$ algorithm to the k shortest simple paths algorithm. That is, the algorithm outputs k different simple $s-t$ paths, where the length of the kth path is an α-approximation to the length of the actual kth shortest simple $s-t$ path.

The k shortest simple path problem itself has many applications, several of which are listed in [6]. The most common application ([6], [12]) is when we need to find an $s-t$ path that is short, but which also satisfies several other constraints. If the constraints are complicated enough, the resulting optimization problem might take too long to solve directly, so instead we could find several different short paths

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

and choose the one that best satisfies the constraints. A natural choice would be to compute the k shortest simple paths, but this could take a while, so it might be better to compute the *approximate* k shortest simple paths.

Another important application of replacement paths comes from auction theory; they allow us to compute the *Vickrey Prices* of edges that are owned by selfish agents in a network. We do not go into detail, but intuitively, the value of an edge e for some shortest path $\pi(s,t)$ depends on the difference between the original length of $\pi(s,t)$, and the length of the shortest $s-t$ path avoiding e. This is exactly what replacement paths allow us to compute. See Nisan and Ronen [14] or Hershberger and Suri [9] for a more complete overview.

1.2 Existing Algorithms As mentioned before, the fastest algorithm for replacement paths in general graphs has a running time of $O(mn + n^2 \log\log n)$, and the fastest algorithm for k shortest simple paths has a running time of $O(k(mn+n^2 \log\log n))$. Moreover, Hershberger *et al.* proved a $\Omega(m\sqrt{n})$ lower bound for both these problems in the path comparison model.

However, many efficient algorithms are known in special classes of graphs. We mention fastest running times for replacement paths, but the fastest running times for k shortest simple $s-t$ paths are just a factor of k larger. Malik *et al.* [13] show that in *undirected* graphs, replacement paths can be computed in only $O(m + n \log n)$ time (See also Hershberger and Suri [9]). Roditty and Zwick [16] presented an algorithm for directed, *unweighted* graphs that finds all replacement paths in $\widetilde{O}(m\sqrt{n})$ time. Finally, Emek *et al.* [5] presented an $O(n \log^3(n))$ algorithm for finding replacement paths in weighted, directed *planar* graphs (Klein *et al.* later developed an $O(n \log^2(n))$ algorithm [11]).

There are no previous approximation algorithms for replacement paths, but Roditty [15] presented an approximation algorithm for the k shortest simple $s-t$ paths in general graphs. The stretch is $3/2$, and the running time is $O(k(m\sqrt{n}+n^{3/2}\log n))$. This is an important result because it showed the *possibility* of using approximation to improve the running time. This possibility was by no means a given; for example, Dor *et al.* [3] showed that approximation *cannot* help us beat the o(mn) bound for all pairs shortest paths in weighted, directed graphs.

1.3 Our Contributions We present the first approximation algorithm for replacement paths; our algorithm works in directed graphs with positive edge weights. Given any $\epsilon \in [0,1)$, we present a $(1+\epsilon)$ approximate algorithm with a running time of $O(\epsilon^{-1}\log^2 n \log(nC/c)(m + n\log n)) = \widetilde{O}(m \log(nC/c)/\epsilon)$, where C is the largest edge weight in the graph and c is the smallest weight. We also present an even faster algorithm for the simpler problem of computing a $(1+\epsilon)$ approximation to the second shortest simple $s-t$ path – the running time is $O(\epsilon^{-1}\log^2 n(m + n\log n)) = \widetilde{O}(m/\epsilon)$. By Roditty's reduction in [15], this yields a $(1+\epsilon)$ approximate algorithm for finding the k shortest simple paths in $O(k\epsilon^{-1}\log^2 n(m + n\log n)) = \widetilde{O}(km/\epsilon)$ time. Note that all of our running times are nearly optimal, except for the $\log(nC/c)$ factor in the replacement paths algorithm.

Moreover, our algorithm beats the $\Omega(m\sqrt{n})$ lower bound for *exact* replacement paths and exact second shortest simple path in directed, weighted graphs. Thus, our algorithm is the first to definitively prove that finding $(1+\epsilon)$ approximate solutions to these problems is easier than finding exact solutions; Roditty's approximation algorithm [15] for the second shortest simple path beat the *best known* exact algorithm, but it did not overcome the $\Omega(m\sqrt{n})$ lower bound.

1.4 Related Work The variant of the k shortest paths problem where we allow paths with cycles turns out to be much easier – Eppstein showed how to find the k shortest *arbitrary* paths in $O(m + n\log n + k)$ time [6].

A generalization of the replacement paths problem is to compute what can be thought of as *all pairs* replacement paths. In particular, we want to build an oracle that given *any* triplet of vertices x, y, v, outputs the length of the shortest $x-y$ path that avoids v. We might also want to avoid an edge instead of a vertex. There are several algorithms for this problem, but the state of the art was developed by Bernstein and Karger [1]. They present an oracle that has a space of $\widetilde{O}(n^2)$, a constant query time, and a $\widetilde{O}(mn)$ construction time. Duan and Pettie [4] show how to construct an oracle for avoiding *two* failures (vertices or edges) – the construction time is polynomial, the space is only $\widetilde{O}(n^2)$, and the query time is constant. Chechick *et al.* [2] showed that if we are willing to settle for approximate distances then we can efficiently handle f failures for any constant f (the approximation ratio is constant but proportional to the number of failures).

1.5 Organization The rest of this paper is organized as follows. The first seven sections focus on

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

computing an approximate second shortest simple path because this algorithm contains all of our main conceptual ideas, but is technically less involved; the approximate replacement paths algorithm is left for the very end.

Section 2 describes our notation and covers a few basic concepts relating to second shortest simple paths. Section 3 presents our basic approach. We successively compute a tighter and tighter upper bound by breaking our algorithm into $O(\log n)$ phases. Each phase is supposed to find the shortest simple path from a certain set of available paths, and each relevant simple path is represented in some phase – thus, our last step is to take the minimum result over all the phases. Of course, we cannot efficiently compute the *exact* shortest path for each phase; instead, we show that in each phase, either we can in fact compute the exact shortest path for this phase, **or** the shortest path from a *previous* phase closely approximates the one for this phase, so we can effectively ignore the current phase. This can be thought of as a generalization of Roditty's algorithm [15], which only uses two phases, and does not guarantee as close of an approximation between phases.

Section 4 describes our approach more formally. For each phase, we create a modification of our original graph that captures a subset of possible second shortest paths; our goal is now to find the shortest $s-t$ path in this modified graph. However, the path returned for the *modified* graph must satisfy an additional constraint which corresponds to returning a *simple* path in the *original* graph. This problem is easy to solve with multiple runs of Dijkstra's algorithm, but we cannot afford multiple runs.

Section 5 overcomes this problem by presenting a new technique called Progressive Dijkstra. We do in fact run Dijkstra's algorithm many times, but we do not start from scratch each time; instead, we rely on information from previous runs. Intuitively, ordinary Dijkstra works by finding a shorter and shorter distance to each vertex. But in Progressive Dijkstra, we only bother exploring a vertex when we find a *significantly* shorter distance to it. So if the shortest distance to vertex u in the 20th run is only *slightly* better than the previous best distance, then we can safely ignore the distance in the 20th run because we are only looking for an approximation. This technique allows us to upperbound the number of times a vertex is explored during the sequence of Dijkstra runs – the distance to a vertex can "significantly" decrease only so many times. The end of section 5 contains pseudocode for our entire algorithm so far (this does not include the slight improvement of section 7).

Section 6 presents a formal analysis of our algorithm, while section 7 shows how we can tweak our algorithm to slightly improve the running time. Section 8 shows how our approximate second shortest path algorithm can be extended to return approximate replacement paths. Section 9 concludes our paper and describes a few related open problems.

2 Preliminaries

Let $G = (V, E)$ be a directed graph with non-negative edge weights, and let $n = |V|$, $m = |E|$. Let s and t be two arbitrary vertices in G between which we wish to compute a second shortest simple path. Given any path P, let $w(P)$ be the length of P. For any vertices x, y let $\pi(x, y)$ be the shortest path from x to y, and let $\delta(x, y) = w(\pi(x, y))$. Finally, let $\pi(s, t) = (s = v_1, v_2, ..., v_q = t)$. We start by computing shortest paths from s so that we know $\pi(s, t)$ and all distances along $\pi(s, t)$.

For any x, y, let $\pi_2(x, y)$ be the second shortest simple path from x to y – that is, the shortest simple path from x to y that is not identical to $\pi(x, y)$. Define $\delta_2(x, y)$ analogously. Given any $\epsilon \in (0, 1]$, our goal is to compute a simple path $\widehat{\pi}_2(s, t)$ from s to t such that letting $\widehat{\delta}_2(s, t)$ be the length of $\widehat{\pi}_2(s, t)$ we have $\delta_2(s, t) \leq \widehat{\delta}_2(s, t) \leq (1 + \epsilon)\delta_2(s, t)$.

For simplicity, we assume for the rest of the paper that q (the number of vertices on $\pi(s, t)$) is a power of 2: it is easy to extend the algorithm to arbitrary values of q. Also, our algorithm will only output an approximate second shortest *distance* rather than the path itself. It is easy to extend our algorithm to output the corresponding path. We now define the notion of a detour.

DEFINITION 2.1. *Let P be a simple path. A simple $u - v$ path $D(u, v)$ is said to be a detour of P if $D(u, v) \bigcap P = \{u, v\}$ and u precedes v on P. Throughout this paper all detours are in reference to the path $P = \pi(s, t)$.*

LEMMA 2.1. *The second shortest path from s to t must have the form $\pi(s, v_i) \circ D(v_i, v_j) \circ \pi(v_j, t)$ where $D(v_i, v_j)$ is a detour (so $j > i$). See Figure 1.*

Proof. We know that $\pi_2(s, t)$ cannot contain all the edges on $\pi(s, t)$, so say that it does not contain some edge (v_j, v_{j+1}). The claim is that the shortest $s - t$ path avoiding this edge has the form above. For note that $\pi_2(s, t)$ must deviate from $\pi(s, t)$ at some point v_i *before* v_{j+1} ($i < j+1$). Moreover, it cannot deviate at two points v_i and $v_{i'}$ ($j + 1 > i' > i$) because it would be faster to just take the original shortest path from s all the way to $v_{i'}$. Similarly, $\pi_2(s, t)$ must merge back with $\pi(s, t)$ at some point v_k *after*

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

v_j, and then just follow $\pi(s,t)$ from v_k to t. Thus, $\pi_2(s,t) = \pi(s,v_i) \circ D(v_i,v_k) \circ \pi(v_k,t)$. See Figure 1.

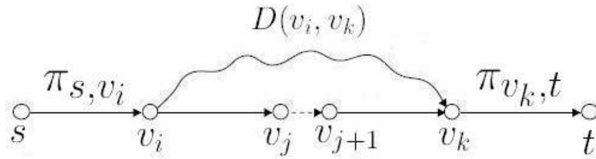

Figure 1: The standard form for a second shortest simple path – see Lemma 2.1. If $\pi_2(s,t)$ avoids the edge (v_j, v_{j+1}) then it has a single detour around that edge.

In finding our approximate second shortest path, we only consider paths of the form in Lemma 2.1. We now characterize paths by their detours.

Definition 2.2. *Define the* span *of a detour* $D(v_i, v_j)$ *to be* $j - i$. *Let the* detour-span *of any path* $\pi(s,v_i) \circ D(v_i,v_j) \circ \pi(v_j,t)$ *be the span of* $D(v_i,v_j)$.

3 Our Approach

In [15], Roditty relies on the fact that it is easy to find the shortest $s-t$ path that has a large detour-span. Thus, he starts by finding the shortest $s-t$ path with a detour-span of at least $\sqrt{n}$ and uses this as an upper bound UB on $\delta_2(s,t)$. He then shows that the shortest path with a smaller detour-span is either easy to find, or it is not that much shorter than the shortest path with large detour-span. In other words, either we can efficiently compute the exact value of $\delta_2(s,t)$, or the upper bound UB serves as a 3/2-approximation to $\delta_2(s,t)$.

Our approach generalizes Roditty's algorithm [15]. Firstly, instead of directly computing a single upper bound we progressively improve our upper bound over a series of $O(\log q)$ phases (recall: q is the number of vertices on $\pi(s,t)$). In particular, the ith phase tries to compute the shortest $s-t$ path with detour-span in $[q/2^i, q/2^{i-1}]$. We then show that either we can efficiently compute the exact shortest path of the ith phase, or the shortest path of some previous phase $j < i$ is a $(1+\epsilon)$ approximation to the shortest path for phase i.

It is crucial that the multiplicative error of each phase is only $(1+\epsilon)$ (rather than 3/2 as in [15]). Not only does this improve the overall stretch factor, but it is also necessary for the improved running time. The problem is that the multiplicative errors of the $O(\log q)$ phases might blow up to an overall error of $(1+\epsilon)^{\log q}$. But whereas $(3/2)^{\log q}$ is huge, our error is tiny because we can choose $\epsilon' = \epsilon/\log q$, leading to an overall error of $(1+\epsilon/\log q)^{\log q} = 1 + O(\epsilon)$.

Definition 3.1. *Let* W_i *be the length of the shortest* $s-t$ *path with detour-span in* $[q/2^i, q/2^{i-1}]$ *(as before,* q *is the number of vertices on* $\pi(s,t)$*). Let* U_i *be the length of the shortest* $s-t$ *path with detour-span* $\geq q/2^i$ *(so* $U_i = \min_{j \leq i}\{W_j\}$*). Note that* $U_{\log q} = \delta_2(s,t)$.

Lemma 3.1. *Let* $\epsilon' = \epsilon/(2\log q)$. *Say that we can construct an algorithm which returns* $O(\log q)$ *values* $R_1, ..., R_{\log q}$ *that satisfy the following properties.*

1. $\delta_2(s,t) \leq R_1 \leq U_1$
2. *If* $U_i < U_{i-1}/(1+\epsilon')$ *then* $\delta_2(s,t) \leq R_i \leq U_i$
3. *Else,* $\delta_2(s,t) \leq R_i$

Then, letting $R = \min_i\{R_i\}$, *we have that* $\delta_2(s,t) \leq R \leq (1+\epsilon)\delta_2(s,t)$. *That is,* R *is a suitable approximation to the second shortest distance.*

Remark 3.1. *The lemma above spells out our approach. The algorithm will run in phases, with phase* i *outputting* R_i. *It may seem strange that in properties 1 and 2, instead of just requiring* $R_i = U_i$, *we allow for the possibility of* $R_i \leq U_i$. *The reason is that in phase* i *we consider* all *paths with detour-span in* $[q/2^i, q/2^{i-1}]$, *but in doing so we also happen to consider a few paths with smaller detour-span.*

Proof. Let k be the largest index for which $U_k < U_{k-1}/(1+\epsilon')$ (if no such index exists set $k=1$) and note that by property 2, $\delta_2(s,t) \leq R_k \leq U_k$ (if k = 1 we rely on property 1 instead). We now claim that $\delta_2(s,t) \leq R_k \leq (1+\epsilon)\delta_2(s,t)$. For by our definition of k we know that $U_i \geq U_{i-1}/(1+\epsilon')$ for any $i > k$. But this means that

$$U_k \leq (1+\epsilon')U_{k+1} \leq (1+\epsilon')^2 U_{k+2} \leq ... \\ \leq (1+\epsilon')^{\log q} U_{\log q} = (1+\epsilon/(2\log q))^{\log q}\delta_2(s,t)$$

Which yields

$$\delta_2(s,t) \leq R_k \leq U_k \leq (1+\epsilon/(2\log q))^{\log q}\delta_2(s,t) \\ \leq (1+\epsilon)\delta_2(s,t) \text{(see footnote)}^1$$

But if R_k is a $(1+\epsilon)$ approximation to $\delta_2(s,t)$ then $R = \min_i\{R_i\} \leq R_k$ is also a $(1+\epsilon)$ approximation to $\delta_2(s,t)$ (property 3 ensures that $R \geq \delta_2(s,t)$), which completes the proof.

4 The Setup

We now describe an algorithm that satisfies the properties of Lemma 3.1. During each phase, we

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

want to consider all paths with certain detour-spans (see Definition 2.2). Our general approach is to label some vertices on $\pi(s,t)$ as *start vertices* and the rest as *finish vertices*. We will then show how to find the shortest $s-t$ path whose detour starts at a start vertex and ends at a finish vertex that succeeds the start vertex on $\pi(s,t)$. Then, we split each phase into a constant number of sub-phases, and assign labellings to the sub-phases in such a way that for any detour of the appropriate span, there is a sub-phase labeling for which this detour starts at a start vertex and ends at a finish vertex.

Given any labeling, we modify the edges of G in a way that makes it easier to find the shortest path whose detour starts at a start vertex and ends at a finish vertex. We remove the incoming edges to every start vertex and the outgoing edges from every finish vertex. We also remove all the edges on $\pi(s,t)$ to avoid just using the original shortest path. Finally, for every start vertex v_i we add an edge from s to v_i with weight $\delta(s,v_i)$. Similarly, for every finish vertex v_j we add an edge from v_j to t with weight $\delta(v_j,t)$. Note that any $s-t$ path of the form $\pi(s,v_i)\circ D(v_i,v_j)\circ\pi(v_j,t)$ (see Lemma 2.1) is represented in our new graph as long as v_i is a start vertex and v_j is a finish vertex – we take the edge $(s,\, v_i)$, then detour $D(v_i,v_j)$, and then $(v_j,\, t)$.

This suggests a very simple algorithm for computing an R_1 with $\delta_2(s,t)\le R_1\le U_1$ (see Definition 3.1). We let all v_i for $i\le q/2$ (recall: q is the number of vertices on $\pi(s,t)$) be start vertices and we let the rest be finish vertices; we then modify the graph accordingly. It is easy to see that any detour with span $\ge q/2$ must start at a start vertex and end at a finish vertex. Thus, computing the shortest distance from s to t in our modified graph yields the desired R_1. The running time is $\widetilde{O}(m)$ – the cost of a single run of Dijkstra's algorithm from s.

Unfortunately, it is not always this easy. Let us focus on constructing a labeling for phase i – the phase in which we compute the shortest path with detour-span in $[q/2^i, q/2^{i-1}]$. We split $\pi(s,t)$ into intervals of size $q/2^i$. So the first $q/2^i$ vertices go into the first interval, the next $q/2^i$ vertices go into the second interval, and so on. Let $I_1,...,I_{2^i}$ be the resulting intervals.

We now split the phase into four sub-phases. In sub-phase 1 we label all the vertices in $I_1,I_5,I_9,I_{13},...$ as start vertices and the rest as finish vertices; we refer to $I_1,I_5,...$ as the start intervals. More generally, the start intervals of sub-phase j are $I_j,I_{j+4},I_{j+8},...$ (see Figure 2). It is easy to see that for any v_j,v_k with $k-j\in[q/2^i,q/2^{i-1}]$, we have that v_j and v_k are either one or two intervals apart. Either way, in the sub-phase where v_j is a start vertex, v_k must be a finish vertex, so every path with detour-span in $[q/2^i,q/2^{i-1}]$ is represented in one of the four sub-phases.

We now want to compute shortest paths from s in each sub-phase. Unfortunately, we cannot naively run Dijkstra from s because our path must satisfy the requirement that the start vertex *precedes* the finish vertex on $\pi(s,t)$. Running Dijkstra from s might return a path that takes the edge from s to a start vertex v_{90}, then takes some path to a finish vertex v_{10} and then takes the edge to t. Such a path is not allowed because it corresponds to a non-simple path in the original graph: the path from $s=v_1$ to v_{90}, then back to v_{10}, and then from v_{10} to $v_q=t$ (see Figure 2).

Before we present our algorithm for computing shortest paths whose start vertex precedes the finish vertex on $\pi(s,t)$, let us summarize what we have so far

DEFINITION 4.1. *Given any labeling L of the vertices on $\pi(s,t)$ (into start and finish vertices) let $\delta(L)$ be the length of the shortest path of the form $\pi(s,v_i)\circ D(v_i,v_j)\circ\pi(v_j,t)$, where v_i is some start vertex, v_j is some finish vertex, and $j>i$.*

DEFINITION 4.2. *Let $L_{i,j}$ be the labeling described above for the jth sub-phase of phase i. That is, we split $\pi(s,t)$ into 2^i intervals of size $q/2^i$, and then label the vertices in every fourth interval (starting with I_j) as the start vertices, and the rest as finish vertices. Finally, let L_1 be the labeling we described for computing U_1 in phase 1: the first $q/2$ vertices are start vertices, and the rest are finish vertices.*

LEMMA 4.1. *Say that we present an algorithm which for any sub-phase j of any phase i outputs a value $R_{i,j}$ with the following properties (for $i=1$ the algorithm only outputs a single value R_1).*

1. $R_1=\delta(L_1)$.
2. *If $\delta(L_{i,j})<U_{i-1}/(1+\epsilon')$ then $R_{i,j}=\delta(L_{i,j})$.*
3. *If $\delta(L_{i,j})\ge U_{i-1}/(1+\epsilon')$ then our only requirement is $R_{i,j}\ge\delta(L_{i,j})$.*

Then: the values $R_i=\min_j\{R_{i,j}\}$ satisfy the requirements of Lemma 3.1.

Proof. We already argued that $\delta(L_1)\le U_1$, so if property 1 is satisfied in this lemma, it is also satisfied in Lemma 3.1. Similarly, property 3 of Lemma 3.1 is satisfied because every $R_{i,j}$ is clearly $\ge\delta_2(s,t)$. Finally, to see that property 2 of Lemma 3.1 is

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

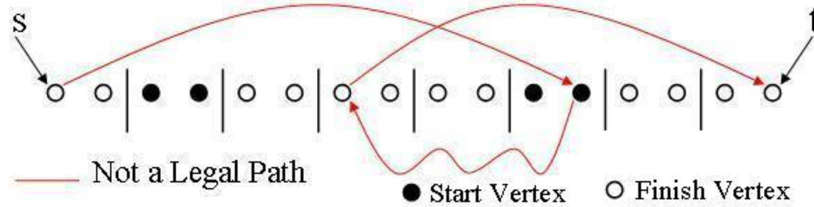

Figure 2: An example of a labeling for sub-phase $j = 2$ of phase $i = 3$, with $q = 16$. There are $2^i = 8$ intervals of size $q/2^i = 2$. Every fourth interval is a start interval, starting with the second because we are in sub-phase 2. Although not shown, there is an edge from s to every start vertex and from every finish vertex to t. We want to find the shortest path from s to t *for which the start vertex precedes the finish vertex on* $\pi(s,t)$.

satisfied, let us focus on some i for which $U_i < U_{i-1}/(1+\epsilon')$. It is clear that since $U_i < U_{i-1}$ we must have $W_i = U_i$ (see Definition 3.1). Thus, U_i is the length of some simple $s - t$ path P that has a detour-span in $[q/2^i, q/2^{i-1}]$.

But we know that every path with such a detour-span is represented in one of the sub-phases of phase i. That is, there is some sub-phase k for which P's detour starts at a start vertex of labeling $L_{i,k}$ and ends at a finish vertex of $L_{i,k}$. But $\delta(L_{i,k})$ is the *shortest* distance with this property so we must have

$$U_{i-1}/(1+\epsilon') > U_i = w(P) \geq \delta(L_{i,k})$$

This completes the proof because it shows that $U_i < U_{i-1}/(1+\epsilon')$ implies $\delta(L_{i,k}) < U_{i-1}/(1+\epsilon')$, so by property 2 of this lemma we have $R_{i,k} = \delta(L_{i,k}) \leq U_i$, so $R_i = \min_j\{R_{i,j}\} \leq R_{i,k} \leq U_i$.

Note that we have already presented an algorithm for computing $\delta(L_1)$ with just a single run of Dijkstra's algorithm, so we ignore L_1 from now on.

5 Progressive Dijkstra

Our goal is now to present an efficient algorithm for computing $\delta(L_{i,j})$. We mentioned before that we cannot handle all start vertices at once because this might lead to the finish vertex preceding the start vertex on $\pi(s,t)$. Our solution is to split the algorithm into many stages, each of which handles a single start interval. So in sub-phase 1, stage 1 would find the shortest $s - t$ path whose detour starts at I_1. Then, before proceeding to I_5, we would delete all the vertices on $\pi(s,t)$ that precede I_5 (in order to to prevent cycles) – stage 2 can now safely compute the shortest path whose detour starts at I_5. Before proceeding to I_9 we would delete all the vertices before I_9, and so on. The problem with this approach is that it requires multiple runs of Dijkstra's algorithm: one for each stage.

We overcome this by using a *progressive* version of Dijkstra. The intuitive description is that when running Dijkstra in stage 3 we do not just start from scratch: we use the information from stages 1 and 2. If the distance to some vertex u in stage 3 is larger than in one of the previous stages then there is no reason for us to explore u in stage 3. In fact, since we are looking for an approximation, we only have to explore u if the distance in stage 3 is *significantly* smaller than in all previous stages. This upper bounds the number of times we will have to explore a vertex u; the distance to u can only "significantly" decrease so many times.

We now give a formal description. See Figure 3 at the end of this section for pseudocode describing our entire second-shortest path algorithm. We focus on finding $\delta(L_{i,j})$ for some arbitrary $L_{i,j}$. Let $I_1, I_2, ..., I_{2^i}$ be our intervals of size $q/2^i$, and recall that in sub-phase j the start intervals are $I_j, I_{j+4}, I_{j+8}, ...$ As before, we delete all edges on $\pi(s,t)$, we delete edges entering start vertices or leaving finish vertices, and we add edges from every finish vertex to t. But we do not yet add edges from s to the start vertices since we do not handle all start vertices at once. Instead, our algorithm runs in stages: the kth stage handles the kth start interval (interval I_{4k+j-4}).

Now, we could start each stage from the source s. But the problem with s is that it serves two roles – it is our source, but it is also a regular vertex on $\pi(s,t)$ that could have up to $O(n)$ edges. We cannot afford to look at $O(n)$ edges per stage, so *we create a new source s'*, leaving s to be treated as an ordinary vertex on $\pi(s,t)$. Now, instead of adding an edge of weight $\delta(s, v_i)$ from s to every vertex v_i

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

in the kth start interval, we will add an edge from s' to v_i of weight $\delta(s, v_i)$. Note that the weight will still depend on the original s because even if s' is our new source, we still care about the distance from s to t. Intuitively, we can view s and s' as basically interchangeable in our algorithm – the only difference is that there may be some edges (s, u) in the *original* graph which we do not replicate for s'. We now proceed with our description of Progressive Dijkstra.

Just as in ordinary Dijkstra, we store a value $c(u)$ for each vertex u – this represents the shortest distance to u we have found so far. We start with $c(s') = 0$ and $c(u) = \infty$ for every $u \neq s'$. Note that these values are not reset during every stage: $c(u)$ represents the shortest distance we have found *among all stages so far*.

In stage k, we delete all the vertices on $\pi(s, t)$ that precede the kth start interval and we add an edge of weight $\delta(s, v_i)$ from s' to every v_i in the kth start interval. We now run a modified version of Dijkstra from s'. Instead of exploring *every* vertex (as in ordinary Dijkstra), we only explore a vertex u if we changed the value of $c(u)$ during the current stage. To ensure that we do not explore u too often, we set a high threshold for changing $c(u)$. **When we relax an edge (u', u), we only change $c(u)$ to $c(u') + w(u', u)$ if $c(u') + w(u', u) < c(u)/(1 + \epsilon')$** ($\epsilon' = \epsilon/(2 \log q)$ – see Lemma 3.1).

Actually, it is a bit more complicated; we do not always have a high threshold for changing $c(u)$. When we relax (u', u), we only use the high-threshold relax procedure if $c(u)$ has not yet been changed *during this stage*. Otherwise, if $c(u)$ has already been changed in the current stage (*i.e.* the high threshold was satisfied earlier in the stage), we use the regular relax procedure for u; that is, we decrease $c(u)$ if $c(u') + w(u', u) < c(u)$. The reasoning behind this is that no matter how many times we change $c(u)$ in a given stage, we explore u at most once in that stage because Dijkstra only explores each vertex once. *So in any stage, either we do not explore u at all, or we explore it exactly once, in which case c(u) must have decreased by at least a $(1 + \epsilon')$ factor* (since the initial threshold is high).

(Technical note: for edges that enter t we *always* use the regular relax procedure. So when relaxing (u,t), we would decrement $c(t)$ even if $c(u) + w(u, t) = c(t) - 1$. We can do this safely because the whole point of having a high threshold for changing $c(u)$ was to ensure that we do not explore u too often. But t is our final destination so we never explore it anyway, so we can change $c(t)$ as often as we like).

All in all, each stage runs a modification of Dijkstra's algorithm (with a Fibonacci heap – see [7]), but some vertices are not explored even if they have the smallest current value; if that value is from a previous stage then we can ignore it. After the last stage, we output $R_{i,j} = c(t)$. Again, Figure 3 contains pseudocode for our whole algorithm.

6 Analysis

6.1 Running Time We start by analyzing the running time for computing $\delta(L_{i,j})$ using Progressive Dijkstra (see Definition 4.2). Each vertex u only gets explored in a stage if $c(u)$ decreases by at least a $(1 + \epsilon')$ factor (recall: $\epsilon' = \epsilon/(2 \log q)$). Letting C be the largest edge weight in the graph, and c the smallest positive edge weight, it is easy to see that each vertex is explored at most $O(\log_{(1+\epsilon')}(nC/c)) = O((1/\epsilon') \log(nC/c)) = O(\epsilon^{-1} \log(n) \log(nC/c))$ times. This yields a running time of $\widetilde{O}(m \log(nC/c))$ for computing $\delta(L_{i,j})$. There are $O(\log n)$ phases and a constant number of subphases for each, so there are $O(\log n)$ labellings $L_{i,j}$, so the total running time is $\widetilde{O}(m \log(nC/c))$. More precisely, it is $O((m + n \log n)\epsilon^{-1} \log^2 n \log(nC/c))$. Section 7 shows how we can shave off the $\log(nC/c)$, but even what we have now is quite efficient.

6.2 Correctness We now prove that the values $R_{i,j}$ satisfy the properties of Lemma 4.1. We focus on some arbitrary labeling $L_{i,j}$, and start by noting that Progressive Dijkstra clearly only considers paths represented by this labeling – paths whose detour starts at a start vertex, and ends at a finish vertex that succeeds the start vertex. Thus, we will always return $R_{i,j} \geq \delta(L_{i,j})$.

Definition 6.1. *Given a simple path P containing vertices x, y with x preceding y, let $\delta[P](x, y)$ be the distance from x to y along the path P. That is, $\delta[P](x, y)$ is the length of the subpath of P that goes from x to y.*

Let $P_g = \pi(s, v_g) \circ D(v_g, v_h) \circ \pi(v_h, t)$ be the shortest path in labeling $L_{i,j}$ – that is, $w(P_g) = \delta(L_{i,j})$. Now, if $h - g \geq q/2^{i-1}$ then P_g is a path with detour-span $\geq q/2^{i-1}$, so $U_{i-1} \leq w(P_g) = \delta(L_{i,j})$ (see Definition 3.1 for U_{i-1}), so we are only concerned with property 3 of Lemma 4.1. This is trivially satisfied because as we discussed, we always return $R_{i,j} \geq \delta(L_{i,j})$.

Thus, we assume that $h - g < q/2^{i-1}$. Now, let $D(v_g, v_h) = (v_g = w_1, w_2, w_3, ..., w_r = v_h)$, and say that v_g belongs to the kth start interval of our

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Figure 3: Pseudocode for the algorithm presented in sections 3-5. This does not include the improvement of section 7.

Algorithm 1: **Main**($G(V,E)$)

Output: $(1+\epsilon)$ approximation to $\delta_2(s,t)$
foreach $1 \le i \le \log q$ **do**
  $R_i \leftarrow$ **PhaseValue**($G(V,E), i$)
end
$R \leftarrow \min_i\{R_i\}$;
return R

Algorithm 2: **PhaseValue**($G(V,E), i$)

Output: Phase Value R_i
`Break` $\pi(s,t)$ `into intervals`
  $I_1, \ldots I_{2^i}$ `of length` $q/2^i$;
$E_i \leftarrow E - \{$`edges on` $\pi(s,t)\}$;
$E_i \leftarrow E_i - \{$`edges leaving` $t\}$;
foreach $1 \le j \le 4$ **do**
  $R_{i,j} \leftarrow$ **SubSetup**($G(V,E_i), i, j$)
end
$R_i \leftarrow \min_j\{R_{i,j}\}$;
return R_i;

Algorithm 3: **SubSetup**($G(V,E), i, j$)

Output: $R_{i,j}$
`Comment: this procedure sets up the labeling` $L_{i,j}$;
$E_j \leftarrow E$;
`Label` $I_j, I_{j+4}, I_{j+8}, \ldots$ `as start intervals` ;
`Label the rest as finish intervals`;
foreach v *in a start interval* **do**
  delete all edges $(\cdot, v)$ from E_j
end
foreach v *in a finish interval* **do**
  1. delete all edges $(v, \cdot)$ from E_j;
  2. add an edge (v,t) of weight $\delta(v,t)$ to E_j ;
end
$R_{i,j} \leftarrow$ **SubValue**($G(V,E_j), i, j$);
return $R_{i,j}$

Algorithm 4: **SubValue**($G(V,E), i, j$)

Output: The value $R_{i,j}$
$E_k \leftarrow E$;
$V_k \leftarrow V$;
`Create new vertex` s' ;
`Create a new function` c `with`
  $c(s') = 0$ `and` $c(v) = \infty \ \forall v \in V$
foreach $1 \le k \le 2^i/4$ **do**
  $I \leftarrow$ k-th start interval ;
  foreach v *preceding* I *on* $\pi(s,t)$ **do**
    delete v from V_k
    (and all incident edge from E_k)
  end
  foreach $v \in I$ **do**
    1. add edge (s', v) with weight $\delta(s,v)$ to E_k ;
    2. **ProgDijk**($G(V_k, E_k), s', c$)
  end
  ;
end
return $c(t)$

Algorithm 5: **ProgDijk**($G(V,E), s', c$)

Input: c is a function on the vertices
`Comment: there is no output but the function` c `changes.`
`Comment: We create an initially empty Fibonacci Heap H`
insert$(H, s', 0)$
insert$(H, t, c(t))$
while $H \neq \emptyset$ **do**
  $u \leftarrow$ **Extract-Min**(H)
  For every edge (u,u') **Relax**(u, u', c, H)
end

Algorithm 6: **Relax**(u, u', c, H)

Input: c is a function on the vertices ;
H is a Fibonacci heap of vertices ;
`Comment:` $\epsilon' = \epsilon/2\log q$;
`Comment: there is no output but` c `and` H `change.`
if $u' \in H$ **and** $c(u') > c(u) + w(u,u')$ **then**
  $c(u') \leftarrow c(u) + w(u,u')$
  Insert($H, u', c(u')$)
end
if $u' \notin H$ **and**
  $c(u') > (1+\epsilon')(c(u) + w(u,u'))$ **then**
  $c(u') \leftarrow c(u) + w(u,u')$
  Decrease-Key($H, u', c(u')$)
end

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

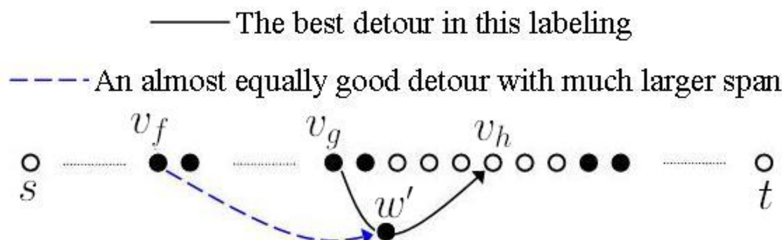

Figure 4: A figure proving the correctness of Progressive Dijkstra. Either we find the best detour for the labeling (the black detour from v_g to v_h), or there exists an almost equally good detour with a large span (the blue path from v_f to w' followed by the black path from w' to v_h).

labeling $L_{i,j}$. Then, in stage k, as Dijkstra progresses, we relax the edge (s', v_g), then later $(v_g, w_2), (w_2, w_3)$, and so on until (w_{r-1}, v_h), and finally (v_h, t). One option is that we end up exploring every w_i. This would mean that we explore every vertex on the path P_g, and so clearly end with $R_{i,j} = c(t) = w(P_g) = \delta(L_{i,j})$, as desired.

But what if we do not explore all of P_g? Let w' be the first vertex on $D(v_g, v_h)$ for which $c(w')$ did not change in this stage. By definition of our relax procedure, this could only happen if $c(w') \leq (1+\epsilon')\delta[P_g](s, w')$. But this value of $c(w')$ must have been set in a previous stage. In particular, there must be some $s-t$ path P_f detouring from a start vertex v_f in a previous start interval (see Figure 4), for which

$$\delta[P_f](s, w') = c(w') \leq (1+\epsilon')\delta[P_g](s, w')$$

But let us now consider the path $P = \pi(s, v_f) \circ D(v_f, v_h) \circ \pi(v_h, t)$, where $D(v_f, v_h)$ follows the path P_f from v_f to w' and then the path P_g from w' to v_h (see Figure 4). We have

$$\begin{aligned} w(P) &= \delta[P_f](s, w') + \delta[P_g](w', t) \\ &\leq (1+\epsilon')\delta[P_g](s, w') + \delta[P_g](w', t) \\ &\leq (1+\epsilon')\delta[P_g](s, t) = (1+\epsilon')\delta(L_{i,j}) \end{aligned}$$

But P has a detour-span greater than $3(q/2^i) > q/2^{i-1}$ because $h - f > g - f$, and v_g, v_f are in different start intervals, so they are separated by at least 3 finish intervals of size $q/2^i$. Thus, $w(P) \geq U_{i-1}$, so we have $U_{i-1} \leq w(P) \leq (1+\epsilon')\delta(L_{i,j})$, so we are only concerned with property 3 of Lemma 4.1, which is trivially satisfied.

All in all, we have shown that either we explore all of P_g and thus compute an exact value for $\delta(L_{i,j})$, or we are only concerned with property 3 of Lemma 4.1 (because $\delta(L_{i,j}) \geq U_{i-1}/(1+\epsilon')$).

7 A Slight Improvement

Although quite efficient, the algorithm presented above has a running time proportional to $O(\log(nC/c))$. In this section, we show how to remove this factor.

In the algorithm above, we only explored a vertex u when $c(u)$ decreased by at least a $(1+\epsilon')$ multiplicative factor. In this section we replace this with an additive threshold. That is, we want to only explore u when $c(u)$ decreases by some *additive* factor α.

To see how this works, say that we are in phase i, and let α be any number in $[0, \epsilon' U_{i-1}/2]$ (we set α later). Now, everywhere in our relax procedure where we required a multiplicative decrease of at least $(1+\epsilon')$, we instead require an additive decrease of a least α. That is, we only change $c(u)$ to $c(u') + w(u', u)$ if $c(u') + w(u', u) < c(u) + \alpha$ (note: our low-threshold relax procedure, where we only require that $c(u') + w(u', u) < c(u)$, remains the same). We now show that the requirements of Lemma 4.1 are still satisfied under this new relax procedure (note: property 2 of Lemma 4.1 retains *its original form*, $(1+\epsilon')$ multiplicative factor and all).

We show this by examining the correctness proof for Progressive Dijkstra. In the original correctness proof we showed that either we can directly find the shortest path in a labeling $L_{i,j}$, or we have $U_{i-1} \leq (1+\epsilon')\delta(L_{i,j})$. With our additive relax procedure we have that either we directly find the shortest path, or we have $U_{i-1} \leq \delta(L_{i,j}) + \alpha \leq \delta(L_{i,j}) + \epsilon' U_{i-1}/2$, which implies $U_{i-1} \cdot (1 - \epsilon'/2) \leq \delta(L_{i,j})$. But some simple algebra shows that since ϵ' is in $[0, 1]$ we have that $(1 - \epsilon'/2) \geq 1/(1+\epsilon')$ (see footnote

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

[2]), so $U_{i-1} \cdot (1-\epsilon'/2) \leq \delta(L_{i,j})$ implies that $U_{i-1}/(1+\epsilon') \leq \delta(L_{i,j})$. In other words, under this new relax procedure, we still have that either we can directly find the shortest path under labeling $L_{i,j}$ or we have $U_{i-1}/(1+\epsilon') \leq \delta(L_{i,j})$ – this is precisely property 2 of Lemma 4.1, so correctness still holds.

The running time analysis is slightly trickier because we do not actually know U_{i-1} (we only know R_{i-1}), so it is unclear what value we should set for α. Say, however, that we knew U_{i-1}, and that in particular, we have already found a simple $s-t$ path (other than $\pi(s,t)$) of length U_{i-1}. Then, we would set $\alpha = \epsilon' U_{i-1}/2$ and add another minor detail to our relax procedure: we do not explore a vertex u unless $c(u) \leq U_{i-1}$. This is perfectly safe because we have no need for distances greater than U_{i-1} (because of our assumption that we have already found a path of length U_{i-1}). But u is only explored when $c(u)$ drops by at least $\alpha = \epsilon' U_{i-1}/2$, so every vertex u is explored at most $O(U_{i-1}/\alpha) = O(2/\epsilon') = O(\log n/\epsilon)$ times.

Of course, the above analysis assumed knowledge of U_{i-1}. But say that instead of knowing U_{i-1} we knew bounds LB, UB with the following properties:

- We have already found a simple $s-t$ path (other than $\pi(s,t)$) of length $\leq UB$.
- $LB \leq U_{i-1}$ and $LB \leq UB$.

Then, notice that we can safely not explore a vertex u unless $c(u) < UB$ (we have no need for distances $\geq UB$ since we have already found one). On the other hand, we can make the additive threshold of our relax procedure $\alpha = \epsilon' LB/2 \leq \epsilon' U_{i-1}/2$. It is easy to see that every u is explored at most $O(\epsilon^{-1}(UB/LB)\log n)$ times. The following lemma shows how we can find valid LB, UB with $UB/LB \leq 2$. This ensures that each vertex is explored at most $O(\log n/\epsilon)$ times (per subphase), which ensures an overall running time of $O((m + n\log n)\epsilon^{-1}\log^2 n)$.

LEMMA 7.1. *Say that we are in phase i (so we have already outputted $R_{i-1}, R_{i-2}, ...R_1$). Then, we can safely set $UB = \min_{i-1 \geq j \geq 1}\{R_j\}$ and $LB = UB/2$.*

Proof. The setting for UB is clearly valid since every R_j is the length of some path we have already found. We must now show that that $UB/2 \leq U_{i-1}$. The proof is nearly identical to that of Lemma 3.1. In that proof we assumed that we were in phase $\log q$, but all of the steps can trivially be extended to the case where $i \neq \log q$. As before, the main idea is that our multiplicative error increases by at most $(1+\epsilon')$ between phases, and so is bounded by $(1+\epsilon')^i \leq (1+\epsilon')^{\log q} \leq (1+\epsilon) \leq 2$.

[2]We show this by noting that for any $x \in [0,1]$ $(1-x/2)(1+x) \geq 1$. For $(1-x/2)(1+x) = 1+x/2-x^2/2 \geq 1+x/2-x/2 = 1$

8 Replacement Paths

We now show how to generalize the above algorithm to find approximate replacement distances for $\pi(s,t)$. That is, for every v_i on $\pi(s,t)$ we compute a $(1+\epsilon)$ approximation to the length of the shortest path from s to t avoiding v_i. Note that this can trivially be extended to avoiding the *edges* of $\pi(s,t)$ – just add a vertex to the middle of each edge on $\pi(s,t)$, and the path avoiding that vertex is the path avoiding the edge. The running time of our algorithm is $O((m + n\log n)\epsilon^{-1}\log^2 n \log(nC/c))$, where C is the largest edge weight on the graph and c is the smallest weight; unlike in our approximate second shortest path algorithm, we do not know how to remove the $\log(nC/c)$. It is not hard to extend our algorithm to return approximate replacement *paths* (or some subset of the paths), although this may increase the running time as it takes $\widetilde{O}(L)$ time to output a path with L edges (details omitted).

Our replacement paths algorithm is somewhat technically complicated, but the concepts used are nearly identical to those of our second shortest simple path algorithm.

DEFINITION 8.1. *Let $\pi(s,t,v_i)$ be the shortest path from s to t avoiding v_i, and let $\delta(s,t,v_i)$ be the length of this path.*

For replacement paths, we slightly change the role of our labelings. We will again group our vertices into intervals, but now, we no longer restrict our detours to ending in finish intervals; the detour must start in a start interval, but in can end in a *different* start interval. In terms of how we decide on start and finish vertices, we use exactly the same labelings $L_{i,j}$ as before (see Definition 4.2).

DEFINITION 8.2. *Given a labeling L (into start and finish vertices), we say that two start vertices are in different intervals if they are separated by at least one finish vertex. The definition is analogous for finish vertices, and a start and a finish vertex are always said to be in different intervals. Note that this definition corresponds to two vertices being in different intervals $I_k, I_{k'}$ in our labelings $L_{i,j}$.*

DEFINITION 8.3. *We say that an $s-t$ path $P(s,t)$ is represented by a labeling L if the detour of $P(s,t)$*

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

starts at a start vertex and ends at a vertex that is not in the same interval, and that succeeds the start vertex on $\pi(s,t)$. Given a vertex v_g on $\pi(s,t)$ we say that a path $P(s,t)$ is represented by (L, v_g) if the detour $D(v_f, v_h)$ of $P(s,t)$ satisfies the following properties: $f < g < h$ (so the path avoids v_g), v_f is a start vertex, and v_f is in a different interval than both v_g and v_h.

DEFINITION 8.4. *We let $\delta(L)$ be the length of the shortest $s-t$ path that is represented by labeling L. We let $\delta(L, v_g)$ be the length of the shortest $s-t$ path that is represented by (L, v_g)*

Since we slightly changed the definition of $\delta(L)$, we modify Progressive Dijkstra a bit. The only difference is that we do not delete the edges entering start vertices (because our detours can end in start vertices). As before, in the kth stage, we delete all vertices before the kth start interval and add an edge (s', v_i) of weight $\delta(s, v_i)$ for every v_i in the kth start interval. We also delete all edges entering any v_i in the kth start interval (except (s', v_i)) – this is to ensure that the detours starting in the kth start interval do not also end in this interval. We then run Dijkstra from s' with our high-threshold relax procedure – we use the multiplicative $(1+\epsilon')$ threshold (the one used in the original description of Progressive Dijkstra, before the improvement of Section 7). The last detail to cover is that we always use the regular (low-threshold) relax procedure for edges leaving s'. So at the end of stage k, we always have $c(v_i) = \delta(s, v_i)$ for every v_i in the kth start interval because when we relax (s', v_i) we necessarily set $c(v_i) = \delta(s, v_i)$ (of course, we then delete v_i in the next stage).

To find the approximate second shortest simple path we used progressive Dijkstra to attempt to find $\delta(L_{i,j})$ for each labeling. In particular, we had a function c on the vertices where $c(u)$ corresponded to the shortest distance to u that we had found so far. We ended by returning $c(t)$, since we only cared about the distance to t.

But when computing replacement paths, we do not just care about a single value $c(t)$ because we want the shortest $s-t$ distances avoiding each of the q vertices on $\pi(s,t)$ individually. Our approach is to note that this information is quite easy to compute by looking at $c(v_i)$ for various v_i on $\pi(s,t)$.

DEFINITION 8.5. *Let $c[i,j](v_g)$ be the value of $c(v_g)$ after we finish running Progressive Dijkstra on labeling $L_{i,j}$ (see Definition 4.2 for $L_{i,j}$). Let $c[i,j,k](v_g)$ be the value of $c(v_g)$ at the end of stage k of Progressive Dijkstra on labeling $L_{i,j}$.*

DEFINITION 8.6. *For every labeling $L_{i,j}$ and vertex $v_g \in \pi(s,t)$, we define a function $m[i,j](v_g)$ as follows. Define index k such that the kth start interval is the last start interval before v_g (if v_g itself is in a start interval, pick the previous start interval). We define*

$$m[i,j](v_g) = \min_{h>g}\{c[i,j,k](v_h) + \delta(v_h, t)\}$$

LEMMA 8.1. *$m[i,j](v_g) = \delta(L_{i,j}, v_g)$ (see Definition 8.4 for $\delta(L_{i,j}, v_g)$).*

Proof. As in the definition of $m[i,j](v_g)$, let the kth start interval be the last start interval before v_g. Also, Let $P(s,t)$ be the shortest $s-t$ path that is represented by $(L_{i,j}, v_g)$ (so $w(P(s,t)) = \delta(L_{i,j}, v_g)$). Let $D(v_f, v_h)$ be the detour of $P(s,t)$. By the definition of represented, we know that v_f is a start vertex that precedes v_g on $\pi(s,t)$, so in particular, v_f is in the kth start interval or in some interval before it. Thus,

$$c[i,j,k](v_h) \le \delta(s, v_f) + w(D(v_f, v_h))$$

and so

$$\begin{aligned} m[i,j](v_g) &\le c[i,j,k](v_h) + \delta(v_h, t) \le w(P(s,t)) \\ &= \delta(L_{i,j}, v_g) \end{aligned}$$

On the other hand, we also know that $m[i,j](v_g) \ge \delta(L_{i,j}, v_g)$ because every term in the min clause of $m[i,j](v_g)$ clearly corresponds to the length of some $s-t$ path that is represented by $(L_{i,j}, v_g)$.

REMARK 8.1. *Lemma 8.1 sets the foundation for how we extend our second shortest simple path algorithm to also find replacement paths. Intuitively, just as previously our final approximation to $\delta_2(s,t)$ was $\min_{i,j}\{R_{i,j}\} = \min_{i,j}\{c[i,j](t)\}$, our final approximation to $\delta(s,t,v_g)$ will be $\min_{i,j}\{m[i,j](v_g)\}$.*

DEFINITION 8.7. *Given a vertex v_g and a detour $D = D(v_i, v_j)$ with $i < g < j$, let the g-span of D be $g - i$ (note: $g-i$ not $j-i$). Given any path of the form in Lemma 2.1 that avoids v_g, let its detour-g-span be the g-span of its detour.*

DEFINITION 8.8. *Let $W_i(v_g)$ be the length of the shortest $s-t$ path avoiding v_g with detour-g-span in $[q/2^i, q/2^{i-1}]$ (as before, q is the number of vertices on $\pi(s,t)$). Let $U_i(v_g)$ be the length of the shortest $s-t$ path avoiding v_g with detour-g-span $\ge q/2^i$ (so $U_i(v_g) = \min_{j\le i}\{W_j(v_g)\}$). Note that $U_{\log q}(v_g) = \delta(s,t,v_g)$.*

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

LEMMA 8.2. *Say that we have an algorithm that for every $1 \leq i \leq \log q$ and every $v_g \in \pi(s,t)$ outputs a value $R_i(v_g)$ with the following properties.*

1. $\delta(s,t,v_g) \leq R_1(v_g) \leq U_1(v_g)$
2. *If* $U_i(v_g) < U_{i-1}(v_g)/(1+\epsilon')$ *then* $\delta(s,t,v_g) \leq R_i(v_g) \leq U_i(v_g)$
3. *Else,* $R_i(v_g) \geq \delta(s,t,v_g)$

Then: letting $R(v_g) = \min_i\{R_i(v_g)\}$ *we have* $\delta(s,t,v_g) \leq R(v_g) \leq (1+\epsilon)\delta(s,t,v_g)$.

Proof. The proof is exactly the same as for Lemma 3.1

LEMMA 8.3. *Any $s-t$ path P with detour-g-span $\geq q/2^i$ is represented by $(L_{i,j}, v_g)$ for some j, so in particular, $w(P) \geq \delta(L_{i,j}, v_g)$ (see Definition 8.3 for represented).*

Proof. Let the detour of P be $D(v_f, v_h)$. We know that $h - f > g - f \geq q/2^i$, so since intervals only have size $q/2^i$, we must have that v_f is in a different interval from both v_g and v_h. Moreover, we know that v_f is a start vertex in one of the $L_{i,j}$, so P is represented in that $(L_{i,j}, v_g)$.

LEMMA 8.4. *Say that we have an algorithm that for every $1 \leq i \leq \log q$, $1 \leq j \leq 4$, and every $v_g \in \pi(s,t)$ outputs a value $R_{i,j}(v_g)$ with the following properties (the algorithm only outputs a single value $R_1(v_g)$).*

1. $R_1(v_g) = \delta(L_1, v_g)$
2. *If* $\delta(L_{i,j}, v_g) < U_{i-1}(v_g)/(1+\epsilon')$ *then* $R_{i,j}(v_g) = \delta(L_{i,j}, v_g)$
3. *Else,* $R_{i,j}(v_g) \geq \delta(L_{i,j}, v_g)$

Then: the values $R_i(v_g) = \min_j\{R_{i,j}(v_g)\}$ satisfy the properties of Lemma 8.2.

Proof. Since every path with detour-g-span in $[q/2^i, q/2^{i-1}]$ is represented by some $(L_{i,j}, v_g)$ (see Lemma 8.3), there must be some k such that $\delta(L_{i,k}, v_g) \leq U_i(v_g)$. The proof now continues exactly as the proof of Lemma 4.1.

(We omit the details but efficiently computing $R_1(v_g) = \delta(L_1, v_g)$ for every v_g is very easy. Thus, we only focus on computing appropriate $R_{i,j}$ values.)

LEMMA 8.5. *The values $R_{i,j}(v_g) = m[i,j](v_g)$ satisfy the properties of Lemma 8.4.*

Proof. Properties 1 and 3 are clearly satisfied. To see that property 2 holds, say that the start interval right before v_g is the kth start interval (if v_g itself is in a start interval, we are focusing on the *previous* start interval). Now, let P_f be the shortest $s-t$ path avoiding v_g that is represented by $(L_{i,j}, v_g)$ – that is, $w(P_f) = \delta(L_{i,j}, v_g)$. Let $P_f = \pi(s, v_f) \circ D(v_f, v_h) \circ \pi(v_h, t)$ where $f < g < h$ and v_f is in the k_fth start interval with $k_f \leq k$. Now, if during the k_fth stage of Progressive Dijkstra we end up exploring all of $D(v_f, v_h)$, then we have $c[i,j,k](v_h) \leq c[i,j,k_f](v_h) = \delta[P_f](s, v_h)$ (see Definition 6.1 for $\delta[P_f]$). But then note that

$$\begin{aligned} m[i,j](v_g) &\leq c[i,j,k](v_h) + \delta(v_h, t) \\ &\leq \delta[P_f](s, v_h) + \delta(v_h, t) \\ &= \delta[P_f](s,t) = \delta(L_{i,j}, v_g) \end{aligned}$$

Thus, $R_{i,j}(v_g) = m[i,j](v_g) \leq \delta(L_{i,j}, v_g)$, so property 2 is satisfied.

But what if we do not end up exploring all of $D(v_f, v_h)$ in stage k_f? Then, by the same argument as in the correctness proof for finding the second shortest simple path (Section 6.2), there must be some path $P_b = \pi(s, v_b) \circ D(v_b, v_h) \circ \pi(v_h, t)$ such that v_b is in a start interval $k_b < k_f$ and

$$\begin{aligned} \delta(L_{i,j}, v_g) \leq w(P_b) &\leq (1+\epsilon')w(P_f) \\ &= (1+\epsilon')\delta(L_{i,j}, v_g) \end{aligned}$$

But since v_b is in an earlier start interval than v_f we have $f - b \geq 3q/2^i > q/2^{i-1}$, so v_b has detour-g-span $\geq q/2^{i-1}$, so $U_{i-1}(v_g) \leq w(P_b)$. Thus, $U_{i-1}(v_g) \leq (1+\epsilon')\delta(L_{i,j}, v_g)$, so we are not concerned with property 2 of Lemma 8.4, so we are done because property 3 is trivially satisfied.

We are basically done. All we have left to show is how to compute the $m[i,j](v_g)$. As before, say that the last start interval before v_g is the kth start interval. We could, at the end of stage k, just compute $m[i,j](v_g) = \min_{h>g}\{c[i,j,k](v_h) + \delta(v_h, t)\}$ by looking at every v_h. The problem is that this could take $O(n)$ time, and we cannot afford to spend $O(n)$ per vertex v_g. We fix this by using a simple data structure to help us compute the min clause quickly.

As we run Progressive Dijkstra on labeling $L_{i,j}$ the function c changes. We create an array A of length q where

$$A[h] = c(v_{q-h}) + \delta(v_{q-h}, t)$$

Note that the entries of this array change as the function c changes. Our goal is to create a data structure which given any input g at any time during our execution of Progressive Dijkstra outputs the

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

value

$$\min_{h<g} A[h] = min_{h>q-g}\{c(v_h)+\delta(v_h,t)\} = m[i,j](v_{q-g})$$

Then, we can compute $m[i,j](v_g)$ by giving input $q-g$ at the end of the kth stage.

We present a data structure for this problem that has query and update times of $O(\log n)$. So every time $c(v_i)$ changes for any v_i we must spend $O(\log n)$ time updating the structure. But recall that because of our high-threshold relax procedure, each $c(v_i)$ changes at most $O(\log_{(1+\epsilon')}(nC/c)) = O(\epsilon^{-1}\log(n)\log(nC/c))$ times (technical note: actually, $c(v_i)$ can change many times because recall that once the high-threshold relax procedure is satisfied during a stage, we use the regular relax procedure for the rest of the stage. So although it is true that $c(v_i)$ can only change in a small number of stages, it can change many times within a single stage. However, this complication is trivial to avoid by only updating the array A *at the very end of every stage*; we never query A in the middle of a stage.)

Thus, the total update time for this labeling is $O(n\epsilon^{-1}\log^2(n)\log(nC/c))$, which leads to an update time of $O(n\epsilon^{-1}\log^3(n)\log(nC/c))$ over all phases, which is within our desired bounds. It is not hard to check that the total query time throughout all phases is only $O(n\log^2 n)$.

The Data Structure: For every pair of integers $0 \le i \le \log q$ and $1 \le j \le q/2^i$ we let $B[i,j]$ be the subarray of A between elements $A[j2^i]$ and $A[(j+1)2^i]$. It is easy to check the following three properties of the arrays $B[i,j]$

1. Their total size is $O(n\log n)$
2. Every $A[k]$ belongs to at most two arrays $B[i,\cdot]$ (for every i), so it belongs to $O(\log n)$ of the B arrays.
3. For any k, the subarray $A[1], A[2], ..., A[k]$ is the union of $O(\log n)$ of the $B[i,j]$. This is because there is clearly some B array which gets us at least half way to k, the another which covers at least half the remaining distance, and so on.

We let $p[i,j]$ be the minimum value in $B[i,j]$ – by property 1, we can find all the $p[i,j]$ in $O(n\log n)$ time. Whenever we change a value in A, we change $O(\log n)$ of the $B[i,j]$ arrays, and we must update the corresponding $p[i,j]$. But note that the values in A only decrease (because $c(u)$ only decreases as Progressive Dijkstra executes), so we can trivially maintain $p[i,j]$ in $O(1)$ time per change to $B[i,j]$. Thus, our update time is $O(\log n)$. For the query we rely on property 3 – to find the minimum value of $A_k = A[1], ..., A[k]$, we just use the minimum values in the $O(\log n)$ B arrays whose union is A_k. This completes the description of the data structure, and hence of our approximate replacement paths algorithm.

9 Conclusion

We have presented $(1+\epsilon)$ approximate algorithms for computing replacement paths and k shortest simple paths in weighted, directed graphs. The running times are $\widetilde{O}(mlog(nC/c)/\epsilon)$ and $\widetilde{O}(mk/\epsilon)$ respectively. The main open question is whether we can find the *exact* second shortest simple path in $O(n^{3-\delta})$ time for any constant $\delta > 0$? Alternatively, can we improve upon the $\Omega(m\sqrt{n})$ lower bound [10] in the path comparison model? Finally, can we improve upon the $\widetilde{O}(m\sqrt{n})$ upper bound for finding the exact second shortest simple path in *unweighted* graphs? It would also be nice to remove the dependence on $O(\log(nC/c))$ in our approximate replacement paths algorithm.

References

[1] A. Bernstein and D. Karger, *Improved distance sensitivity oracles via random sampling*, in Proc. of the 19th SODA, San Francisco, California, 2008, pp. 34–43.

[2] S. Chechick, M. Langberg, D. Peleg, and L. Roditty, *f-sensitivity distance oracles*, Unpublished Manuscript, (2009).

[3] D. Dor, S. Halperin, and U. Zwick, *All-pairs almost shortest paths*, SIAM J. Comput., 29 (2000), pp. 1740–1759.

[4] R. Duan and S. Pettie, *Dual-failure distance and connectivity oracles*, in Proc. of the 20th SODA, New York, New York, 2009, pp. 506–515.

[5] Y. Emek, D. Peleg, and L. Roditty, *A near-linear time algorithm for computing replacement paths in planar directed graphs*, in Proc. of the 19th SODA, San Francisco, California, 2008, pp. 428–435.

[6] D. Eppstein, *Finding the k shortest paths*, SIAM J. Comput., 28 (1999), pp. 652–673.

[7] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. ACM, 34 (1987), pp. 596–615.

[8] Z. Gotthilf and M. Lewenstein, *Improved algorithms for the k simple shortest paths and the replacement paths problems*, Inf. Process. Lett., 109 (2009), pp. 352–355.

[9] J. Hershberger and S. Suri, *Vickrey prices and shortest paths: What is an edge worth?*, in Proc. of the 42nd FOCS, Las Vegas, Nevada, USA, 2001, pp. 129–140. Erratum in FOCS '02.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

[10] J. Hershberger, S. Suri, and A. Bhosle, *On the difficulty of some shortest path problems*, ACM Trans. Algorithms, 3 (2007), p. Article no. 5.

[11] P. Klein, S. Mozes, and O. Weimann, *Shortest paths in directed planar graphs with negative lengths: a linear-space o(n log2 n)-time algorithm*, in Proc. of the 19th SODA, New York, New York, 2009, pp. 236–245.

[12] E. Lawler, *A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem*, Management Science, 18 (1972), pp. 401–405.

[13] K. Malik, A. K. Mittal, and S. K. Gupta, *The k most vital arcs in the shortest path problem*, Operations Research Letters, 4 (1989), pp. 223–227.

[14] N. Nisan and A. Ronen, *Algorithmic mechanism design*, Games and Economic Behavior, 35 (2001), pp. 166–196.

[15] L. Roditty, *On the k-simple shortest paths problem in weighted directed graphs*, in Proc. of the 18th SODA, New Orleans, Louisiana, 2007, pp. 920–928.

[16] L. Roditty and U. Zwick, *Replacement paths and k simple shortest paths in unweighted directed graphs.*, in Proc. of the 32nd ICALP, Lisboa, Portugal, 2005, pp. 249–260.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.