Improved Distance Sensitivity Oracles Via Random Sampling

Aaron Bernstein* David Karger†

Abstract

We present improved oracles for the distance sensitivity problem. The goal is to preprocess a graph G = (V,E) with non-negative edge weights to answer queries of the form: what is the length of the shortest path from x to y that does not go through some *failed* vertex or edge f. There are two state of the art algorithms for this problem. The first produces an oracle of size $\widetilde{O}(n^2)$ that has an O(1) query time, and an $\widetilde{O}(mn^2)$ construction time. The second oracle has size $O(n^{2.5})$, but the construction time is only $\widetilde{O}(mn^{1.5})$. We present two new oracles that substantially improve upon both of these results. Both oracles are constructed with randomized, Monte Carlo algorithms. For directed graphs with non-negative edge weights, we present an oracle of size $\widetilde{O}(n^2)$, which has an O(1) query time, and an $\widetilde{O}(n^2\sqrt{m})$ construction time. For unweighted graphs, we achieve a more general construction time of $\widetilde{O}(\sqrt{n^3 \cdot APSP} + mn)$, where APSP is the time it takes to compute all pairs shortest paths in an aribtrary subgraph of G.

1 Introduction

1.1 The Problem In the *distance sensitivity problem*, we wish to construct a data structure (called an oracle) for a graph G = (V,E) with m edges, n vertices, and non-negative edge weights. The oracle should support the following queries:

• Given vertices (x,y,v), return the length of the shortest path from x to y that avoids v.

• Given vertices (x,y,u,v), return the length of the shortest path from x to y that avoids edge (u,v).

We may also want to extend the oracle to support the corresponding path queries. In this paper, we only show how to answer distance queries avoiding a failed vertex, although all of our oracles can easily be extended to answer the other queries, without increasing the space or time parameters. Path queries take O(L) time, where L is the length of the output path.

There are two main motivations for this problem. The first is modeling a network where vertices (or edges) occasionally fail. When a vertex fails, we don't want to stall distance queries while we recompute shortest paths. With a sensitivity oracle, we can continue answering queries quickly, while constructing a new oracle (for the graph with a vertex deleted) in the background. Constructing a new oracle is rather time consuming, but this is fine as long as failures are relatively rare. The second motivation is Vickrey pricing [10]. In Vickrey pricing, we want to determine how much an edge is worth by calculating the shortest path from x to y without that edge. If we want to find Vickrey prices for many paths at once, then sensitivity oracles are the fastest option.

1.2 Previous Work The naive approach to this problem is to remove each vertex, one at a time, and compute all pairs shortest paths on the resulting graphs. The results are stored in a table of size $O(n^3)$. The construction time is O(n·APSP), where APSP is the time it takes to compute all pairs shortest paths. The best non-trivial oracles were developed by Demetrescu *et al* [6]. Their first oracle has a construction time of $\widetilde{O}(mn^2)$, which is no better than that of the naive oracle. The space, however, is only $O(n^2\log(n))$. Their second oracle only requires $\widetilde{O}(mn^{1.5})$ construction time, but its size is $O(n^{2.5})$. Both oracles answer queries in O(1) time.

The main idea behind the first oracle of Demetrescu *et al* [6] is that in addition to storing information about shortest paths that exclude a single vertex, it also excludes whole sets of vertices at the same time. For every pair (x,y), it designates O(log(n)) intervals on the shortest path from x to y, and then computes the shortest path from x to y that avoids each of these intervals. It can then use this information to return the shortest distance avoiding any failed vertex. The problem with this approach is that it does not reuse information between pairs: every pair requires at least one shortest path computation, which is where the $\widetilde{O}(mn^2)$ construction time comes from.

1.3 Our Contributions Our approach fixes this problem in two ways. Firstly, we pick intervals that lie on multiple shortest paths at once, which allows us to reuse information. It is difficult to deterministically pick such intervals, so we use random sampling. But

*Massachusetts Institute of Technology; Cambridge, MA, 02139; email: bernstei@gmail.com

†Computer Science and Artificial Intelligence Laboratory; Massachusetts Institute of Technology; Cambridge, MA, 02139; email: karger@mit.edu

more importantly, we show that it is not always necessary to directly exclude the intervals by removing them from G, and computing shortest paths. There is other information we can store about these intervals, which still allows the oracle to answer queries efficiently. This information is often much easier to compute.

Our approach significantly improves upon the $\widetilde{O}(\mathrm{mn}^2)$ construction time. Note that we construct our oracles with randomized, Monte Carlo algorithms, so they work with high probability. For directed graphs with non-negative edge weights, we present an oracle of size $O(n^2\log(n))$ that has an $O(1)$ query time, and an $O(n^2\cdot\log(n)\cdot\log^*(n)\cdot\sqrt{S(m,n)} + n\cdot\log^2(n)\cdot S(m,n))$ construction time. S(m,n) is the time it takes to compute single-source shortest paths in an arbitrary weighted, directed graph. Dijkstra's algorithm with Fibonacci heaps [7] achieves $S(m,n) = O(m + n\log(n))$. For unweighted graphs, we achieve a more general construction time of $O(\sqrt{n^3\cdot log^2(n)\cdot APSP} + n\cdot\log^2(n)\cdot S(m,n))$, where APSP is the time it takes to compute all pairs shortest paths in an arbitrary sub-graph of G.

Both algorithms result in an oracle of size $O(n^2 log(n))$, but one advantage of the $\widetilde{O}(n^2\sqrt{m})$ oracle is that it only requires $O(n^2 log^2(n))$ construction space, while the other oracle requires $\widetilde{O}(\min\{n^{3.5}/\sqrt{APSP}, n^4/m\})$. To make for a more intuitive description, we lead up to our final results with a few less efficient results.

2 Notation

We use the same notation as Demetrescu *et al* [6]. Let G = (V,E) be the graph in question. By (u,v), we denote the edge from u to v, if it exists. Like other papers in this field, we assume W.L.O.G that shortest paths are unique, since we can always add small fractional weights to break any ties. Let T_x be the shortest path tree in G rooted at x, and let $\pi_{x,y}$ be the unique shortest path from x to y. Let $\widehat{G}$ be the graph G, only with the edges reversed. $\widehat{T}$ and $\widehat{\pi}$ are the equivalents of T and π for $\widehat{G}$. Note that since shortest paths are unique, $\pi_{x,y}$ contains the same edges as $\widehat{\pi}_{y,x}$, and for any v in $\pi_{x,y}$, both $\pi_{x,v}$ and $\pi_{v,y}$ are subpaths of $\pi_{x,y}$. Let $|\pi|$ denote the number of vertices on the path π, let $h_{x,y} = |\pi_{x,y}|$, and let $d_{x,y}$ denote the length of $\pi_{x,y}$. Also, let $\pi_{x,y,S}$ be the shortest path from x to y that avoids the set of nodes S, and let $d_{x,y,S}$, $h_{x,y,S}$ be the weighted and unweighted lengths of $\pi_{x,y,S}$. For simplicity, we write $\pi_{x,y,\{v\}}$ as $\pi_{x,y,v}$. In any rooted tree T, we say that a node v is at level L if the path from the root to v contains L edges. Let $L_x(L)$ be the set of nodes at level L in T_x, and let $B_x(L)$ be the ball (i.e. set) of nodes at level $\leq$ L.

Let a,b be vertices on $\pi_{x,y}$. We say that a < b if a comes before b on $\pi_{x,y}$. Assuming a $\leq$ b, let the interval [a,b] be the set of vertices v such that a $\leq$ v $\leq$ b. We can now introduce the notion of a *detour*. As far as we know, this notion is at the heart of every algorithm for computing shortest paths with node or link failures. Say that we are given u $\leq$ v on $\pi_{x,y}$, and that we want to find $\pi_{x,y,[u,v]}$. Intuitively, this new path follows $\pi_{x,y}$ for a while, then deviates at some vertex a < u, and then merges back with $\pi_{x,y}$ at some vertex b > v (see figure 1). Thus, we have:

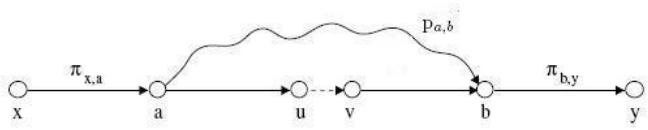

Figure 1: A detour avoiding the interval [u,v] on $\pi_{x,y}$

DEFINITION 2.1. *Let $x \leq a \leq b \leq y$ be nodes on $\pi_{x,y}$. The path $p_{a,b}$ is a detour if $p_{a,b} \bigcap \pi_{x,y} = \{a,b\}$*

LEMMA 2.1. *Any path $\pi_{x,y,[u,v]}$ can be decomposed into three subpaths $\pi_{x,a} \circ p_{a,b} \circ \pi_{b,y}$, where $\circ$ is the path concatenation operator, and $p_{a,b}$ is a detour such that $p_{a,b} = \pi_{a,b,[u,v]}$.*

Proof. For a formal proof, see claim 1.1 of Demetrescu *et al* [6]. As mentioned before, we need to deviate from $\pi_{x,y}$ at some vertex a < u, and then merge back at b > v. Moreover, we will not deviate at both a < u and a < a' < u, since it would be better to just take the subpath $\pi_{x,a'}$ (of $\pi_{x,y}$), and then deviate from a'. Similarly, we only merge back at one vertex b > v. Thus, we have $\pi_{x,y} = \pi_{x,a} \circ p_{a,b} \circ \pi_{b,y}$, and since we want the shortest detour, we must have $p_{a,b} = \pi_{a,b,[u,v]}$.

3 An Overview of Existing Techniques

Our oracles use two techniques developed by Demetrescu *et al* [6]. The first technique allows us to store our distance information more compactly, while the second one speeds up preprocessing.

3.1 Path Cover One of the main difficulties we have to overcome is that it might take $O(n^3)$ space to naively store $d_{x,y,v}$ for all triplets (x,y,v). The solution is to store $d_{x,y,I}$ for various intervals I. We now present a simple lemma which shows how we can use $d_{x,y,I}$ to find $d_{x,y,v}$, for some v $\in$ I.

LEMMA 3.1. *Let $x \leq s < v < t \leq y$ be vertices on $\pi_{x,y}$, where v is our failed vertex. Then, $d_{x,y,v} = min\ \{\ d_{x,s} + d_{s,y,v},\ d_{x,t,v} + d_{t,y},\ d_{x,y,[s,t]}\ \}$.*

Proof. Note that this is just a special case of claim 2.2 in [6]. There are three cases to consider: $\pi_{x,y,v}$ either diverges from $\pi_{x,y}$ after s, merges before t, or avoids the interval [s,t] altogether (see figure 2). Each of these cases is represented by a term in the min clause.

3.2 Excluding Paths Given some source x, and a failed node v, we can compute $d_{x,y,v}$ $\forall$ y $\in$ V in $\widetilde{O}(m)$ time by doing a single-source shortest path computation on the graph G - {v}. We can similarly exclude a whole interval [s,t] in $\widetilde{O}(m)$ time. But this seems wasteful because removing an interval might only affect small portions of T_x, in which case we would like to avoid examining all of G.

Demetrescu *et al* [6] formalize this idea. Say that we are calculating distances from a source x. Let π be any path in T_x that we want to exclude. Define $T_x(\pi)$ to be the subtree of T_x rooted at the first node of π. Note that removing π only affects vertices in $T_x(\pi)$, because if y $\notin$ $T_x(\pi)$, then we must have $d_{x,y,\pi} = d_{x,y}$. Thus, instead of calculating shortest paths on G - $\{\pi\}$, we would like to only explore vertices in $T_x(\pi)$, and edges incident upon those vertices. This can be done by slightly modifying Dijkstra's algorithm (see section 2 of Demetrescu *et al* [6]).

Note that for a given vertex x, and any L, all the vertices at level L in T_x have disjoint subtrees. Thus, since we only have to examine vertices in the subtree of a failed node, we can exclude all vertices at level L (one at a time) in just $\widetilde{O}(m)$ time (we end up exploring each vertex at most once). So in $\widetilde{O}(m)$ time, we can compute $d_{x,y,v}$ $\forall$ y $\in$ V, v $\in$ $L_x(L)$. This proves

LEMMA 3.2. *Recall that $B_x(L)$ is the set of all nodes in T_x that are at level $\leq L$. Given any source x, and a length L, we can exclude all of the vertices in $B_x(L)$, one at a time, in just $\widetilde{O}(mL)$ time. That is, we can compute $d_{x,y,v}$ $\forall y \in V, v \in B_x(L)$.*

3.3 Range Maximum Queries The range maximum data structure is a well known data structure that is used in many of our algorithms. Given an array A, the goal is to construct an oracle call RMQ, such that given a pair of indices (i,j), where i $\leq$ j, RMQ(i,j) returns the maximum element in the subarray A[i], A[i+1], A[i+2], ..., A[j]. This oracle can be built with only linear space and preprocessing time, and it can answer queries in constant time [2]. Range maximum structures can also be used to optimally answer LCA (least common ancestor) queries on rooted trees. The space and preprocessing is once again linear, and queries can be answered in constant time ([2],[9]).

4 Admissible Functions

Many of our results depend on the notion of admissible function, which is new to this paper.

DEFINITION 4.1. *A function $F_{x,y,[s,t]}$ is admissible if $\forall$ $v \in [s,t]$, we have $d_{x,y,[s,t]} \geq F_{x,y,[s,t]} \geq d_{x,y,v}$.*

DEFINITION 4.2. *An important example of an admissible function is $M_{x,y,[s,t]} = max_{v \in [s,t]} \{d_{x,y,v}\}$.*

$M_{x,y,[s,t]}$ differs from $d_{x,y,[s,t]}$ because instead of removing the whole interval [s,t], we just remove the vertices in [s,t] one at a time, and take the longest resulting distance. Note that if we already know $d_{x,y,v}$ $\forall$ v $\in$ [s,t], then $M_{x,y,[s,t]}$ only takes $O(h_{s,t})$ time to compute, whereas $d_{x,y,[s,t]}$ takes $\widetilde{O}(m)$ time.

LEMMA 4.1. **The Triple Path Lemma:** *Let $x \leq s < v < t \leq y$ be vertices on $\pi_{x,y}$ (v is the failed vertex), and let $F_{x,y,[s,t]}$ be an admissible function. Then, $d_{x,y,v}$ = min $\{$ $d_{x,s}$ + $d_{s,y,v}$, $d_{x,t,v}$ + $d_{t,y}$, $F_{x,y,[s,t]}\}$.*

Proof. By definition, since v $\in$ [s,t], $F_{x,y,[s,t]} \geq d_{x,y,v}$. Thus, since the other two terms in the min clause are also $\geq d_{x,y,v}$, we have $d_{x,y,v} \leq$ min $\{$ $d_{x,s} + d_{s,y,v}$, $d_{x,t,v}$ + $d_{t,y}$, $F_{x,y,[s,t]}$ $\}$. But we also know that $F_{x,y,[s,t]} \leq d_{x,y,[s,t]}$, so min $\{$ $d_{x,s} + d_{s,y,v}$, $d_{x,t,v} + d_{t,y}$, $F_{x,y,[s,t]}$ $\}$ $\leq$ min $\{$ $d_{x,s} + d_{s,y,v}$, $d_{x,t,v} + d_{t,y}$, $d_{x,y,[s,t]}$ $\}$ = $d_{x,y,v}$.

5 Random Sampling and Centers

DEFINITION 5.1. *We say that a vertex x covers a vertex v if we store $d_{x,y,v}$ for every y in $T_x(v)$. That is, x covers v if we store shortest distances from x avoiding v.*

A key element of our approach is that we randomly sample *centers*, which store more information than ordinary vertices. For example, say that centers cover every vertex. Then, to find $d_{x,y,v}$, we start by finding a center to the left of v (on $\pi_{x,v}$), and a center to the right. We can now compute the first two terms of the triple path lemma because the centers cover v. The problem is that if centers cover *every* vertex, then we can only afford to have a small number of them. This is fine for paths with many vertices, but if $h_{x,v}$ or $h_{v,y}$ are small, then we might not be able to find centers to the right and left of v. Note, however, that we do not actually need centers that cover every vertex: we just need to find centers that cover v. So if $h_{x,v}$ is small, it suffices to find a center on $\pi_{x,v}$ that covers all vertices in a small ball around it (figure 2). Thus, we have different types of centers: the rare ones cover large balls, while the common ones only cover small balls.

But what about the third term in the triple path lemma? Since we want to conserve space, we can only

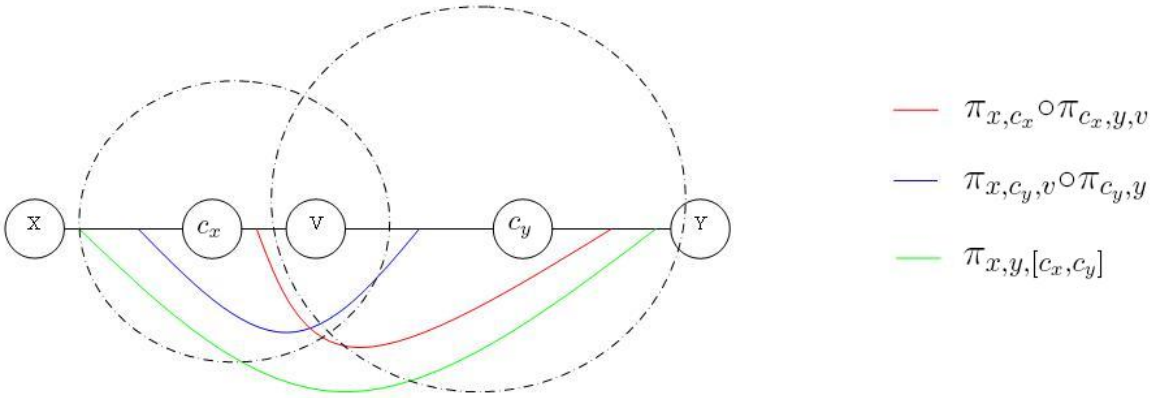

Figure 2: Using centers to apply the triple lemma

afford to store $F_{x,y,I}$ for a small number of intervals I. This collection of intervals needs to span all of $\pi_{x,y}$. This way, if we want to find $d_{x,y,v}$, we can just find an interval I that contains v, and look at $F_{x,y,I}$. But we also need to be able to compute the first two terms of the triple path lemma, so we need the endpoints of our intervals to cover all of the vertices in both of their adjacent intervals. This ensures that if v is in some interval I, the endpoints of I cover v.

Definition 5.2. *A covering chain of $\pi_{x,y}$ is a sequence of vertices c_1, c_2, ..., c_j such that $[c_1, c_j] = \pi_{x,y}$, and c_i covers all the vertices in $[c_{i-1}, c_i]$, and $[c_i, c_{i+1}]$ (see figure 3). If we store $F_{x,y,I}$, for all intervals $[c_i, c_{i+1}]$, then we can use the triple path lemma to efficiently compute $d_{x,y,v}$, for any v.*

We now describe how to arrange for the existence of such covering chains. We have log(n) center *priorities*: centers with low priority are common, but they only cover small balls.

Definition 5.3. *We say that a vertex is a k-center if it has priority k. We define R_k to be the set of k-centers. We say that a k-center c is bigger than some k'-center if $k > k'$. We set $R_1 = V$*

Definition 5.4. *We say that a vertex is a k^+-center if it has priority $\geq k$.*

Sampling: We obtain R_k by sampling each vertex, independently, with probability $\Theta(1/2^k)$.
Center Information: A k-center c covers all vertices in T_c that are not in the subtree of some k+1-center. That is, we move down T_c, covering vertices until we reach a k+1-center.
Covering Chains: Our choice of centers leads to a very natural small covering chain. Given a path $\pi_{x,y}$, we find a list of O(log(n)) centers in ascending priority. So c_1 is the first center on $\pi_{x,y}$, c_2 is the first center bigger than c_1, and so on. Once we get to the biggest center on $\pi_{x,y}$, we begin to descend in priority (see figure 3). It is easy to verify that this is indeed a covering chain.

We store all this information in O(log(n)) lookup tables D_k. D_k[c,y,v] stores $d_{c,y,v}$ if c is a k-center that covers v. We also store $\widehat{D}_k$ for $\widehat{G}$. We now analyze the size and construction time of D_k

Lemma 5.1. *With high probability, any path with at least $2^k log(n)$ vertices contains some k-center. The proof can be found in [8].*

Lemma 5.2. *The following is true with probability $\Omega(1)$: for every center priority k, $|R_k| = O(n/2^k)$. The proof follows from the Chernoff bound [3].*

Lemma 5.3. *Given a k-center c, and a vertex y, c covers $O(2^k)$ vertices on $\pi_{c,y}$ in expectation. By linearity of expectation, there is at least a constant probability that D_k has size $O(n \cdot |R_k| \cdot 2^k)$.*

Lemma 5.4. *With high probability, any k-center c only has to cover vertices that are at level $\leq O(2^k log(n))$ in T_c. This follows from lemma 5.1. By lemma 3.2, this coverage takes $\widetilde{O}(m2^k)$ time.*

Lemmas 5.2 and 5.3 both hold with constant probability, so if we just perform O(log(n)) separate sampling iterations, there is a high probability that both lemmas will hold in one of the iterations. Since this is true with high probability, we just assume that the lemmas always hold. This can be justified with a simple use of the union bound. Combining these lemmas, we get

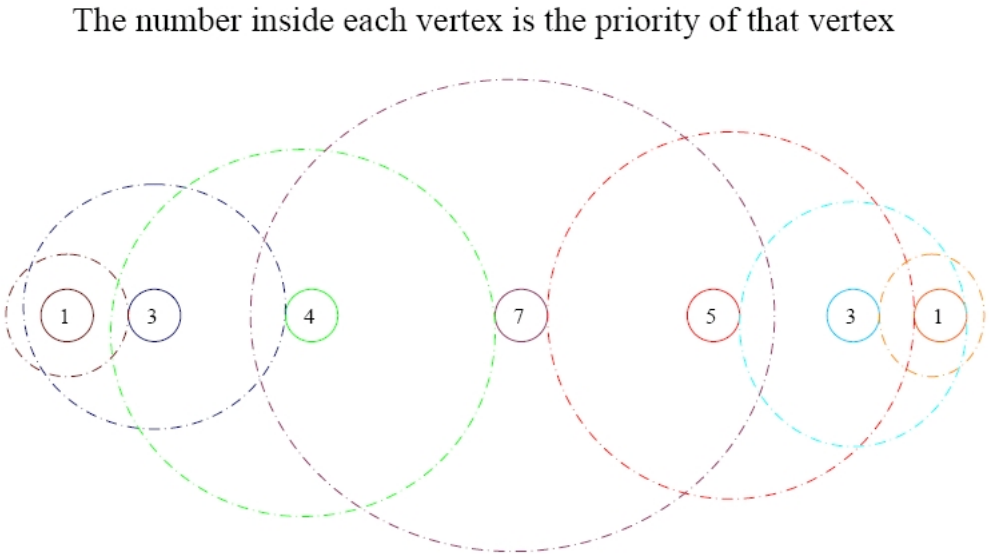

Figure 3: An example of a covering chain that might be used by our oracles

THEOREM 5.1. *We can initialize all the D_k tables in $\widetilde{O}(mn)$ time, and $O(n^2 log(n))$ total space.*

6 The General Framework

All of our oracles use the same general framework. In particular, they store the same lookup tables, and they have the same query procedure. The only difference lies in how they compute the lookup table EP. Below is a list of the tables used in the general framework. We store these tables for both G and $\widehat{G}$. Note that x,y can be any vertices in V, while i,k represent center priorities.

- D[x,y] stores $d_{x,y}$
- H[x,y] stores $h_{x,y}$
- D_k[c,y,v] are the lookup tables from section 5.
- Cr[x,y,i] stores the first i^+-center on $\pi_{x,y}$ (if it exists). Cr stands for center right.
- Cl[x,y,i] stores the first i^+-center on $\widehat{\pi}_{y,x}$ (if it exists). Cl stands for center left.
- BCP[x,y] stores the priority of the biggest center on $\pi_{x,y}$. BCP stands for biggest center priority.
- EP[x,y,i] can store any value that has the following property: let c_x = Cr[x,y,i]. Let c_y be the first center that is larger than c_x on $\pi_{c_x,y}$. If no such center exists, let c_y = Cl[x,y,i]. If this also does not exist, then EP[x,y,i] stores nothing. Note that the intervals $[c_x, c_y]$ corresponding to EP[x,y,·] are precisely the covering chain described in section 5. EP[x,y,i] can store any admissible function $F_{x,y,[c_x,c_y]}$. For example, EP[x,y,i] might store $M_{x,y,[c_x,c_y]}$. EP stands for exclude path.

Query Procedure: Since we already have a covering chain for every pair of vertices, the query procedure is just a simple application of the triple path lemma. In particular, we can use BCP[x,v] to find i: the largest center priority on $\pi_{x,v}$. Given i, we can use Cr and Cl to find the appropriate centers c_x and c_y used in EP[x,y,i]. Both centers cover v, and EP[x,y,i] excludes the interval between them, so we can apply the triple path lemma. Note that the query time is constant. The full query procedure can be found below.

ALGORITHM 6.1. Query Procedure
Input: Three vertices (x,y,v)
Output: $d_{x,y,v}$
If $d_{x,v} + d_{v,y} > d_{x,y}$
    return $d_{x,y}$
i ← BCP[x,v]
j ← BCP[v,y]
If i > j
    break. Compute $\widehat{d}_{y,x,v}$ instead.
c_x ← Cr[x,y,i]
If i=j
    c_y ← Cl[x,y,j]
Else
    c_y ← Cr[v,y,i+1]
d ← $\min\{(d_{x,c_x} + D_i[c_x, y, v]), (d_{c_y,y} + \widehat{D}_j[c_y, x, v])$, EP[x,y,i]}
return d

Space and Preprocessing: The size of the framework is trivially $O(n^2 log(n))$. Preprocessing is more complicated. We can construct D,H, and D_k in $\widetilde{O}(mn)$ time. We can construct Cr, Cl, and BCP in $\widetilde{O}(n^2)$ time by moving down the shortest path tree of each vertex, and

keeping track of the biggest (or the first) center seen on every path. EP is the hardest matrix to initialize. In fact, the rest of this paper describes many different ways of doing so. Note, however, that we have proved:

THEOREM 6.1. *Any algorithm that constructs EP in T time produces a distance sensitivity oracle of size $O(n^2 log(n))$, which has an O(1) query time, and a T + $\widetilde{O}(mn)$ construction time.*

7 A Simple Approach

A useful property of our general framework is that if we somehow manage to compute $d_{x,y,v}$ for all triplets (x,y,v), then in $\widetilde{O}(n^3)$ time, we can compress this information down to $O(n^2 log(n))$ space, instead of $O(n^3)$. All we do is set EP[x,y,i] = $M_{x,y,[c_x,c_y]}$. Since we have already excluded every vertex, this is easy to compute by just taking the maximum of at most n terms. The naive approach to computing $d_{x,y,v}$ (for all triplets) takes $\widetilde{O}(mn^2)$ time, but we can reduce this to $\widetilde{O}(n^3)$ by using the dynamic all pairs shortest path algorithm of Demetrescu and Italiano [4]. There are, however, two drawbacks to this approach. The first is that it is not amenable to any improvement. The second is that even though the size of the oracle is small, the construction space is still $\widetilde{O}(n^3)$.

8 Constructing EP in $\widetilde{O}(mn^{1.5})$ Time

The approach in section 7 works surprisingly well when the i in EP[x,y,i] is small, so that the whole interval $[c_x, c_y]$ is close to x. This is because we can use lemma 3.2 to efficiently exclude a small ball around x. But what about when i is large? In this case, we are only considering intervals that end in a large center. But there are not many large centers, so there are not many intervals we have to exclude. To prove this more formally, we consider the two cases separately. Let c_x, c_y be the endpoints of the interval corresponding to EP[x,y,i]. b is a boundary point to be defined later.

Case 1: i > b

In this case, there are not that many possible intervals $[c_x, c_y]$. By definition, c_x must be the first i^+-center on the shortest path from x to some other vertex. We call such centers *blocking i^+-centers*. c_y must be a bigger (or equally big) center in the subtree of c_x (in T_x). But note that the subtrees of all blocking i^+-centers are disjoint, so each of the $O(|R_b|)$ bigger centers is in the subtree of at most one blocking i^+-center. So in working with x, we only exclude $O(|R_b|)$ intervals. This proves

LEMMA 8.1. *For all $i > b$, we can construct EP[·,·,i] in $\widetilde{O}(mn^2/2^b)$ time.*

Case 2: i ≤ b

In this case, we can brute force the problem. Since i ≤ b, lemma 5.1 tells us that all vertices in $[c_x, c_y]$ are at level ≤ 2^blog(n) in T_x. Thus, for every x, we start by excluding every vertex in $B_x(2^b log(n))$, one at a time. By Lemma 3.2, this only takes $\widetilde{O}(m2^b)$ time. Hence, when computing EP[x,y,i] we already know $d_{x,y,v}$ for every v in $[c_x,c_y]$, so the natural thing to do is set EP[x,y,i] = $M_{x,y,[c_x,c_y]}$. This allows us to compute EP[x,y,i] in just $\widetilde{O}(2^b)$ time by walking down the interval $[c_x,c_y]$ ($[c_x, c_y]$ is small because i is small).

LEMMA 8.2. *For all $i \le b$, we can construct EP[·,·,i] in $\widetilde{O}(mn2^b)$ time. By combining this with lemma 8.1, and setting $b = \sqrt{nlog(n)}$, we can construct EP in $\widetilde{O}(mn^{1.5})$*

9 Constructing EP in $\widetilde{O}(n^3)$ Time in Unweighted Graphs

In the previous oracle, we computed EP by doing about $\sqrt{n}$ shortest path computations per vertex. This is a good approach when m is small, so shortest path computations are cheap. When m is big, however, this becomes expensive. Thus, we show how to use centers to avoid extra shortest path computations. This oracle only requires $\widetilde{O}(n^2)$ construction space.

The basic idea behind this oracle is that instead of just looking at centers on $\pi_{x,y}$, we also look at centers on the optimal path avoiding the failed set in question. For example, if we know that $\pi_{x,y,v}$ contains a k-center, then we know that $d_{x,y,v} = \min_{c \in R_k} \{ d_{x,c,v} + d_{c,y,v} \}$. Thus, our goal is to look at every k-center, and find the shortest replacement path through that k-center.

DEFINITION 9.1. *We say that a path π with h edges is a k-path if $2^k < h \le 2^{k+1}$.*

Given some interval I, say we are told that $\pi' = \pi_{x,y,I}$ is a k-path. By lemma 5.1, we are guaranteed a k' = k - log(log(n)) center on π'. So we only have to examine every k'-center. But given c ∈ $R_{k'}$, how do we quickly find the shortest replacement path through c? Well, the graph is unweighted, so shorter paths must contain fewer edges. Thus, since $h_{x,c,I}$ and $h_{c,y,I}$ are ≤ 2^{k+1}, we must have that $h_{c,x}$ and $h_{y,c}$ are ≤ 2^{k+1}.

So by augmenting every k'-center q (in G and $\widehat{G}$) to cover everything in $B_q(2^{k+1})$, we ensure that c covers every vertex on $\widehat{\pi}_{c,x} = \pi_{x,c}$ and on $\pi_{c,y}$ (see figure 4). Section 9.1 shows how we can use this to compute a replacement path (avoiding I) through c. We omit specific details, but this augmentation does not significantly decrease efficiency. Every k-center q still only covers vertices at level ≤ 2^q (in T_q), so by lemma 3.2, we can still construct D_k in $\widetilde{O}(mn)$ time.

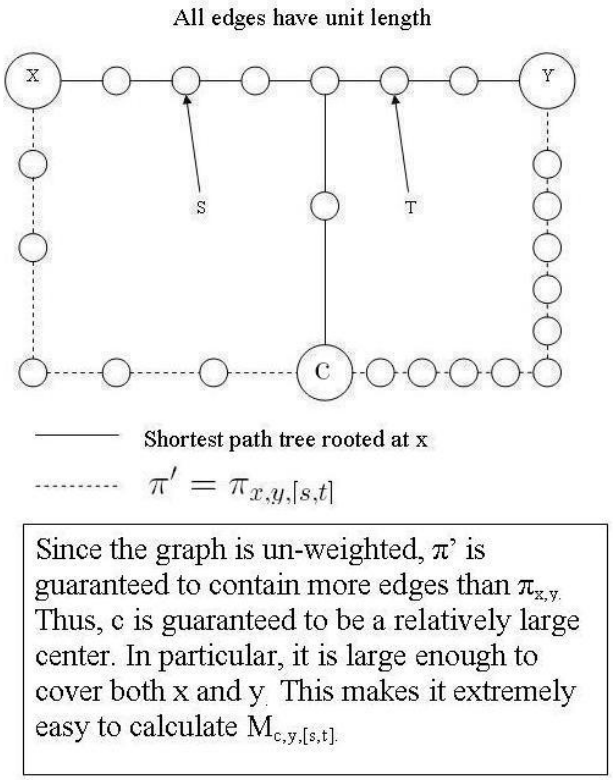

Figure 4: Using centers to quickly compute $M_{c,y,[s,t]}$

In fact, the only drawback to this augmentation is that it requires $O(n^2 \log^2(n))$ construction space. The construction space is slightly worse, but the final oracle still has size $O(n^2 \log(n))$.

Note that this augmentation does not ensure that *every* k'-center covers x and y. It only ensures that every k'-center on π' covers x and y. But this is enough because since a k'-center on π' is bound to exist, we only need to examine k'-centers that cover both x and y.

The final problem is that we assumed that π' is a k-path. But this is not such an unjustified assumption, because k can assume only $O(\log(n))$ values, so we can check all of them. This yields

DEFINITION 9.2. *Let $R_j^{x,y}$ be the set of all j-centers that cover both x (in $\widehat{G}$) and y.*

Key Idea: Let $EP_k = \min_{c \in R_{k'}^{x,y}} \{M_{x,c,[c_x,c_y]} + M_{c,y,[c_x,c_y]}\}$. Note that EP_k captures the intuition we just described, so it finds the shortest replacement path that is a k-path. Thus, we set $EP[x,y,i] = \min_k \{EP_k[x,y,i]\}$. It is not hard to check that this is an admissible function.

9.1 Computing EP_k We now show how given a k'-center c that covers both x (in $\widehat{G}$) and y, we can compute $M_{x,c,[c_x,c_y]} + M_{c,y,[c_x,c_y]}$ in $O(1)$ time. This yields a construction time of $\widetilde{O}(n^2|R_k|)$ for EP_k, and hence $\widetilde{O}(n^2 \Sigma_k |R_k|) = \widetilde{O}(n^3)$ for EP.

We begin with some initial preprocessing. For every k'-center q, and any vertex w that is covered by q, we construct arrays A and B. A[i] is simply the ith vertex on $\pi_{q,w}$, while $B[i] = d_{q,w,A[i]}$ ($d_{q,w,A[i]}$ is available to us because q covers w). Finally, we build a range maximum data structure (section 3.3) on B, which we call $RMQ_{q,w}$. Note that $RMQ_{q,w}(i,j) = M_{q,w,[A[i],A[j]]}$. Since q covers w, we must have $h_{q,w} = O(2^k)$, so we do $O(2^k)$ work for every pair (q,w), so the total preprocessing time is $O(n \cdot 2^k \cdot |R_{k'}|) = \widetilde{O}(n^2)$.

We are almost done. Since c covers y (it is in $R_{k'}^{x,y}$), we can use $RMQ_{c,y}$ to directly compute $M_{c,y,[c_x,c_y]}$. In a first attempt, we let s be the index of c_x (on $\pi_{c,y}$), we let t be the index of c_y, and we compute $RMQ_{c,y}(s,t)$. The only problem is that not all of $[c_x, c_y]$ is necessarily contained in $\pi_{c,y}$, so the index of c_x might not actually exist. Instead, we want s to the index of c': the first node on $[c_x, c_y]$ that is also on $\pi_{c,y}$. Fortunately, s is just $LCA(c, c_x)$ in $\widehat{T}_y$, so we can compute it in constant time (see section 3.3). Thus, we have $M_{c,y,[c_x,c_y]} = \max\{M_{c,y,[c_x,c']}, M_{c,y,[c',c_y]}\}$. The first term of this max clause is just $d_{c,y,c'}$ because no vertex before c' is even on $\pi_{c,y}$. The second term is $RMQ_{c,y}(s,t)$. We have shown how to compute $M_{c,y,[c_x,c_y]}$ in constant time. We compute $M_{x,c,[c_x,c_y]}$ in a similar fashion.

10 Computing EP in $\widetilde{O}(n^3)$ time in weighted graphs

Say we want to compute EP[x,y,i]. Let c_x and c_y be the endpoints of the interval corresponding to EP[x,y,i]. Let $\pi' = \pi_{x,y,[c_x,c_y]}$. Assume, for now, that π' is a k-path (definition 9.1). The reason we cannot directly use the previous algorithm is that we have not accounted for the possibility of long paths with few edges. We still know that π' contains some k'-center c (k' = k - log(log(n))), and it is still the case that $h_{c,y,[c_x,c_y]} \leq 2^{k+1}$. But this no longer implies $h_{c,y} \leq 2^{k+1}$: $\pi_{c,y}$ must be shorter than $\pi_{c,y,[c_x,c_y]}$, but it need not contain fewer edges. Thus, in weighted graphs, c might not cover x and y, so we cannot efficiently compute $M_{c,y,[c_x,c_y]}$ (see figure 5).

In order to fix the above problem, we rely more heavily on the assumption that π' is a k-path. Of course, we do not actually have this knowledge, but we can afford to check every k. Let c be a k'-center on π'. c might not cover y, so we artificially force c to pseudo-cover y, by only covering vertices that are on the shortest path from c to y *that contains at most 2^{k+1} vertices.*

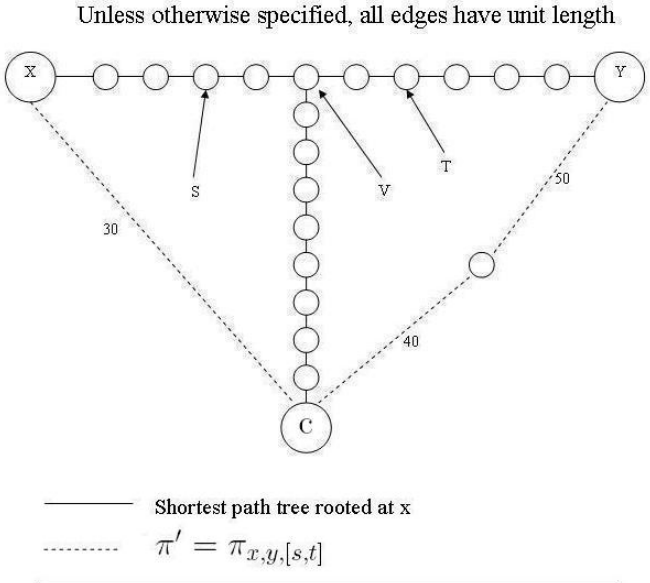

Figure 5: An example of our second oracle failing for a weighted graph

Thus, even if $h_{c,y}$ is big, c only covers 2^{k+1} vertices.

DEFINITION 10.1. *Let $\pi^k_{x,y}$ be the shortest path from x to y that contains at most 2^{k+1} vertices. Let $\pi^k_{x,y,v}$ be the shortest path from x to y that avoids v, and contains at most 2^{k+1} vertices. Define $d^k_{x,y,v}$ and $h^k_{x,y,v}$ analogously.*

At first glance, it seems strange to only cover vertices on $\pi^k_{c,y}$, since $\pi^k_{c,y}$ has almost no relation to $\pi_{c,y}$. Why does our algorithm still work when we restrict ourselves to a seemingly random set of vertices? Fortunately, it is not hard to ensure that EP is an admissible function. To lower bound EP, we exclude all the vertices on $\pi^k_{c,y}$, but we also max the final result with $d^k_{c,y}$. This effectively excludes all of the leftover vertices (on $\pi_{c,y}$) that are not on $\pi^k_{c,y}$, since we are taking a path ($\pi^k_{c,y}$) that does not contain any of them. To upper bound EP, we note that $\pi^k_{c,y}$ is still shorter than $\pi' = \pi_{x,y,[c_x,c_y]}$ because π' is a k-path, so it also contains at most 2^{k+1} vertices.

Thus, our goal is to compute $d_{c,y,v}$ for every v on $\pi^k_{c,y}$. Unfortunately, this is too hard to compute efficiently. Instead, we solve a simpler problem. We find, for every v on $\pi^k_{c,y}$, some value $d^{k-}_{c,y,v}$, such that $d^k_{c,y,v} \geq d^{k-}_{c,y,v} \geq d_{c,y,v}$. In other words, instead of finding the *optimal* replacement path (avoiding v), we just find some replacement path that is better than the optimal replacement path that uses at most 2^{k+1} vertices. This might make EP[x,y,i] larger, but we still have EP[x,y,i] $\leq d_{x,y,[c_x,c_y]}$ because $\pi_{x,y,[c_x,c_y]}$ is itself a replacement path that uses at most 2^{k+1} vertices

So for each center priority k, we create a table SD_k (SD stands for short distance). This table stores $d^{k-}_{c,y,v}$ $\forall$ c $\in R_{k'}$, y $\in$ V, v $\in \pi^k_{c,y}$, where k' = k - log(log(n)). This is similar to D_k, except that we are excluding v $\in \pi^k_{c,y}$ instead of v $\in \pi_{c,y}$, and we are computing $d^{k-}_{c,y,v}$ instead of $d_{c,y,v}$.We show how to construct SD_k in section 10.1

DEFINITION 10.2. *Let $M^{k-}_{x,y,[s,t]} = max_{v \in [s,t] \bigcap \pi^k_{x,y}} \{d^k_{x,y}, d^{k-}_{x,y,v}\}$. This is just like our definition for $M_{x,y,[s,t]}$, except that we are avoiding vertices on $\pi^k_{x,y}$ one at a time, instead of vertices on $\pi_{x,y}$. We also use $d^{k-}_{x,y,v}$ (not $d_{x,y,v}$), and we max with the extra term $d^k_{x,y}$ to ensure that $M^{k-}_{x,y,[s,t]} \geq d^k_{x,y}$.*

Key Idea: We set EP_k[x,y,i] = $\min_{c \in R_{k'}} \{M^{k-}_{x,c,[c_x,c_y]} + M^{k-}_{c,y,[c_x,c_y]}\}$, where k' = k - log(log(n)). We then set EP[x,y,i] = $\min_k \{EP_k[x,y,i]\}$. It is not hard to check that EP is admissible.

10.1 Constructing SD_k Recall that SD_k stores $d^{k-}_{c,y,v}$ $\forall$ c $\in R_{k'}$, y $\in$ V, v $\in \pi^k_{c,y}$, where k' = k - log(log(n)). We describe an algorithm EXCLUDE-SD(c,k), which computes $d^{k-}_{c,y,v}$ $\forall$ y $\in$ V, v $\in \pi^k_{c,y}$. First, we need to compute $\pi^k_{c,y}$ $\forall y \in V$. This can trivially be done in O(m2^k) time, by using the Bellman-Ford algorithm, but stopping after 2^{k+1} iterations [1]. But what about excluding vertices? Just like in section 3.2, when we exclude some vertex v, this only affects a small portion of the vertices in the graph. In particular, by slightly modifying Dijkstra's algorithm, we can restrict our attention to vertices in T_k(v) = {y | v is in $\pi^k_{c,y}$ }. We omit the details because the algorithm we use is very similar to the algorithm EXCLUDE-D used by Demetrescu *et al* (the one presented in section 3.2).

Note that for every vertex y, there are at most 2^k vertices v such that y $\in T_k$(v). Thus, we examine each vertex (and its adjacency list) at most 2^k times, so EXCLUDE-SD(c,k) runs in $\widetilde{O}$(m2^k) time. This leads to an $\widetilde{O}$(mn) construction time, and an O($|R_{k'}| \cdot 2^k \cdot n$) = $\widetilde{O}$(n^2) space requirement for SD_k.

10.2 Constructing EP_k We use similar techniques to the ones used in section 9.1, but they are slightly more complicated. When the graph is unweighted, it is easy to compute $M_{c,y,[c_x,c_y]}$ because the intersection of $[c_x, c_y]$ and $\pi_{c,y}$ is simply a sub-interval of $[c_x, c_y]$. This is because once $\pi_{c,y}$ intersects $[c_x, c_y]$ it will continue

down that interval, since that is the shortest available path. But this is certainly not true of $\pi^k_{c,y}$. $\pi^k_{c,y}$ has an unrelated limitation, so it may actually weave in and out of $[c_x, c_y]$, in an entirely unpredictable manner.

We fix this problem by having a more involved preprocessing stage. Let q be some k'-center, and let w be any vertex. Recall that k' = k - log(log(n)). In section 9.1, we just stored information about $\pi_{q,w}$. The natural parallel for weighted graphs would be to store information about $\pi^k_{q,w}$. But this is not enough because this gives us no information about the relation between $\pi^k_{q,w}$, and possible intervals $[c_x, c_y]$. Thus, we instead store an entire copy of $\widehat{T}_w$, which we call $\widehat{T}^q_w$. We then mark every vertex (in $\widehat{T}^q_w$) that is on $\pi^k_{q,w}$. We also assign a value to each marked vertex, where the value of v is $\mathrm{d}^{k-}_{q,w,v}$. We can do so because this information is stored in $\mathrm{SD}_{k'}$. Unmarked vertices do not have a value.

Now, if we want to find $\mathrm{M}^{k-}_{c,y,[c_x,c_y]}$, we just need to find the maximum value along the path $[c_x, c_y]$ (in the tree $\widehat{T}^q_y$). This is similar to an ordinary range-maximum query, except that we are dealing with a tree instead of an array. Query intervals can start anywhere, but the end point must be in the subtree of the starting point. This is because c_x is always in the subtree of c_y (in $\widehat{T}^q_y$). It is not hard to construct such a data structure by mimicking the techniques used for ordinary range maximum structures. We omit the details, but the preprocessing time is $\mathrm{O}(\mathrm{n}\log^*(\mathrm{n}))$, and queries take $\mathrm{O}(\log^*(\mathrm{n}))$ time.

We can now compute EP_k in the desired time bound. We create $\widetilde{O}(\mathrm{n}\cdot|R_k|)$ trees, each of which can be preprocessed in $\widetilde{O}(\mathrm{n})$ time. This yields a total preprocessing time of $\widetilde{O}(\mathrm{n}^2\cdot|R_k|)$. Once this is done, we can compute $M^{k-}_{x,c,[c_x,c_y]} + M^{k-}_{c,y,[c_x,c_y]}$ in $\mathrm{O}(\log^*(\mathrm{n}))$, so we can construct EP_k[x,y,i] in $\widetilde{O}(|R_k|)$ time.

Note that if we were to actually create (and preprocess) all $\widetilde{O}(\mathrm{n}\cdot|R_k|)$ trees at once, then we would require a construction space of $\widetilde{O}(\mathrm{n}^2|R_k|)$. We fix this problem by working with one k'-center at a time. That is, for all pairs (x,y), we start by computing the shortest replacement paths through some k'-center c. This only requires $\mathrm{O}(\mathrm{n})$ trees, and hence $\widetilde{O}(\mathrm{n}^2)$ space. We then throw out the trees, and move onto another center. As we look at different centers, we store a table of size $\widetilde{O}(\mathrm{n}^2)$ that stores the best replacement paths seen so far

11 Computing EP in $\widetilde{O}(\mathrm{n}^2\sqrt{m})$ time

In sections 9.1 and 10.2, we show how to compute a specific entry EP_k[x,y,i] in $\widetilde{O}(|R_k|)$ time. This is very expensive for small k. Recall, however, that EP_k[x,y,i] corresponds to the case where the replacement path $\mathrm{d}_{x,y,[c_x,c_y]}$ is a k-path. Thus, we present an alternative algorithm which handles small k by efficiently computing replacement paths with few vertices. Combining this with the algorithm in section 10 (which worked well for large k), we get a better construction time. In particular, we introduce a parameter b that serves as our boundary point. To calculate EP_k for k ≥ b, we use the approach in section 10. For k < b we use this new approach. Since k < b, we can assume for the rest of this section that $\mathrm{h}_{x,y,[c_x,c_y]} \leq 2^b$.

The basic idea behind this approach is that we generalize the admissible function $\mathrm{M}_{x,y,[c_x,c_y]}$. In $\mathrm{M}_{x,y,[c_x,c_y]}$, we just excluded every vertex in $[c_x, c_y]$, and took the maximum replacement distance. But say that instead, each vertex v in $[c_x, c_y]$ had a corresponding set S_v that contained v. Then, instead of excluding each vertex, we could exclude the corresponding sets, and take the maximum replacement distance. This approach is useful because multiple vertices could correspond to the same set. Thus, although we cannot afford to exclude every vertex, we can afford to exclude a few large sets whose union contains V.

Definition 11.1. *Given a collection of sets $\mathcal{S}$, let $M^{\mathcal{S}}_{x,y} = max_{S\in\mathcal{S}}\{d_{x,y,S}\}$.*

Definition 11.2. *A collection of sets $\mathcal{S}$ spans an interval I if the union of the sets in $\mathcal{S}$ contains I.*

Lemma 11.1. *If $\mathcal{S}$ is a collection of sets that spans $[c_x, c_y]$, then $M^{\mathcal{S}}_{x,y} \geq d_{x,y,v}$, $\forall\ v \in [c_x, c_y]$.*

By lemma 11.1, it is tempting to just set EP[x,y,i] = $\mathrm{M}^{\mathcal{S}}_{x,y}$. The problem is that we need to somehow upper bound EP. We do this by finding sets S such that $\mathrm{d}_{x,y,S} \leq \mathrm{d}_{x,y,[c_x,c_y]}$. This is not hard to do, because we are assuming that $\mathrm{h}_{x,y,[c_x,c_y]}$ is small ($\leq 2^b$). Thus, a random set S is likely to not intersect $\pi_{x,y,[C_x,C_y]}$. This would immediately imply that $\mathrm{d}_{x,y,S} \leq \mathrm{d}_{x,y,[c_x,c_y]}$.

Definition 11.3. *We say that a collection of sets $\mathcal{S}$ admissibly spans an interval I, if $\mathcal{S}$ spans I, and if for every $S \in \mathcal{S}$, we have $d_{x,y,S} \leq d_{x,y,[c_x,c_y]}$. This trivially yields*

Lemma 11.2. *If $\mathcal{S}$ admissibly spans $[c_x, c_y]$, then EP[x,y,i] = $M^{\mathcal{S}}_{x,y}$ is an admissible function.*

We have reduced our goal to finding a collection of sets that admissibly spans $[c_x, c_y]$. We want to pick sets that don't intersect $\pi_{x,y,[c_x,c_y]}$, but since we don't even know what vertices lie on $\pi_{x,y,[c_x,c_y]}$, the only thing we can really do is pick random sets. We pick $\mathrm{O}(2^b\log(\mathrm{n}))$ random sets, each of which is created by sampling vertices with probability $\Theta(1/2^b)$. Let RS be the collection of random sets. Given any collection $\mathcal{S}$

$\subseteq RS$, we want to be able to quickly compute $M^{\mathcal{S}}_{x,y}$, so we exclude all the sets in RS (one at a time), and compute all pairs shortest paths each time. This takes $\widetilde{O}(2^b \cdot \text{APSP})$ time, where APSP is the time it takes to compute all pairs shortest paths.

DEFINITION 11.4. *Let RS(v) be the set of sets in RS that contain v. By the Chernoff bound [3], there is a high probability that $|RS(v)| \geq log(n)$.*

LEMMA 11.3. *Given $S \in RS$, it is easy to check that there is at least a constant probability that $S \bigcap \pi_{x,y,[c_x,c_y]} = \emptyset$ (because $h_{x,y,[c_x,c_y]} \leq 2^b$). Thus, by the Chernoff bound [3], if we pick O(log(n)) random sets from RS, there is a high probability that one of them does not intersect $\pi_{x,y,[c_x,c_y]}$.*

So for each vertex v on $[c_x, c_y]$, we pick O(log(n)) random sets from RS(v), and then take the one that minimizes $d_{x,y,S}$. Call this set S_v. By lemma 11.3, $d_{x,y,S_v} \leq d_{x,y,[c_x,c_y]}$ (with high probability). Thus, if we let $\mathcal{S}$ be the collection of sets S_v ($v \in [c_x, c_y]$), then $\mathcal{S}$ admissibly spans $[c_x, c_y]$, so by lemma 11.2, we are done. Finding $\mathcal{S}$ takes $\widetilde{O}(h_{c_x,c_y})$ time, so if we can establish some upperbound U on h_{c_x,c_y}, then we can construct EP_k in $\widetilde{O}(2^b \cdot \text{APSP} + n^2 U)$ time (for $k < b$). Recall that for $k \geq b$, we use the algorithm in section 10 to construct EP_k in $\widetilde{O}(n^3/2^b)$ time.

For unweighted graphs, U is just 2^b. This is because shorter paths must contain fewer edges, so since $h_{x,y,[c_x,c_y]} \leq 2^b$, we must have $h_{c_x,c_y} \leq h_{x,y} \leq 2^b$. Setting $b = \log(\sqrt{n^3/APSP})$, we get the desired construction time of $\widetilde{O}(\sqrt{n^3 \cdot APSP} + mn)$. Unfortunately, this upperbound on h_{c_x,c_y} does not hold for weighted graphs. Note, however, that we already have an algorithm for dealing with long paths (section 8). Thus, we use lemma 8.1 to separately take care for the case when $i \geq \log(n) - b$. This takes $\widetilde{O}(mn2^b)$ time, but we can now assume that $i < \log(n) - b$. Because of how we defined EP, the fact that $i < \log(n) - b$ implies that the interval $[c_x, c_y]$ stops once it reaches a $(\log(n) - b)$-center. By lemma 5.1, any shortest path with $n\log(n)/2^b$ vertices contains a $(\log(n) - b)$-center, so $h_{c_x,c_y} \leq n\log(n)/2^b = U$. This yields the desired construction time of $\widetilde{O}(n^2\sqrt{m})$.

Note that if we directly compute all pairs shortest paths for every set in RS, then we need $\widetilde{O}(n^2 2^b)$ construction space. But if we are willing to settle for a construction time of $\widetilde{O}(n^2\sqrt{m})$ (even in unweighted graphs), we can reduce this to $\widetilde{O}(n^2)$ construction space. We just work with one vertex at a time, so that we only have to compute single source shortest paths when excluding RS.

12 Concluding Remarks

We presented a sampling based approach to distance sensitivity oracles which allowed us to significantly decrease construction time, while still preserving a small query time, and a small space requirement. But there is no reason to think that our construction times are optimal; we have made progress towards an $\widetilde{O}(mn)$ construction time for undirected graphs, but we have yet to generalize this to directed graphs. It would also be nice to construct oracles that can handle more than one vertex failure at a time. Finally, any non-trivial lower bounds would be useful.

References

[1] G. Apostolopoulos, R. Gurin, S. Kamat, A. Orda, T. Przygienda, and D.Williams. QoS routing mechanisms and OSPF extensions. Internet Engineering Task Force (IETF), RFC (Experimental) 2676, Aug. 1999.

[2] O.Berkman and U.Vishkin. Recursive star-tree parallel data structure. *SIAM Journal of Computing*, 22(2):221-242, 1993.

[3] H. Chernoff. A measure of the asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Stat*, 23, 493509, 1952.

[4] C. Demetrescu, and G. Italiano. A new approach to dynamic all pairs shortest paths. In *Journal of the ACM*, 51(6):968-992, 2004.

[5] C. Demetrescu and M. Thorup. Oracles for distances avoiding a link-failure. In *Proceedings of 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '92), San Francisco, California*, pages 838 843, 2002.

[6] C. Demetrescu, M. Thorup, R. Alam Chowdhury, and V. Ramachandran. Oracles for distances avoiding a link-failure.

[7] M.L.Fredman and R.E.Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34:596-615, 1987.

[8] D. Greene and D. Knuth. Mathematics for the analysis of algorithms. Birkhauser, Boston, 1982.

[9] D.Harel and R.E.Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338-355, 1984.

[10] J.Hershberger and S.Suri. Vickrey prices and shortest paths: what is an edge worth?. In *Proceedings of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS '01), Las Vegas, Nevada*, pages 129-140, 2001. Erratum in FOCS '02.