A Nearly Optimal Oracle for Avoiding Failed Vertices and Edges

Aaron Bernstein
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA, 02139
bernstei@gmail.com

David Karger
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
77 Massachusetts Avenue
Cambridge, MA, 02139
karger@mit.edu

ABSTRACT

We present an improved oracle for the distance sensitivity problem. The goal is to preprocess a directed 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. The previous best algorithm produces an oracle of size $\widetilde{O}(n^2)$ that has an $O(1)$ query time, and an $\widetilde{O}(n^2\sqrt{m})$ construction time. It was a randomized Monte Carlo algorithm that worked with high probability. Our oracle also has a constant query time and an $\widetilde{O}(n^2)$ space requirement, but it has an improved construction time of $\widetilde{O}(mn)$, and it is deterministic. Note that $O(1)$ query, $O(n^2)$ space, and $O(mn)$ construction time is also the best known bound (up to logarithmic factors) for the simpler problem of finding all pairs shortest paths in a weighted, directed graph. Thus, barring improved solutions to the all pairs shortest path problem, our oracle is optimal up to logarithmic factors.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

1. INTRODUCTION

1.1 The Problem

In the *distance sensitivity problem*, we wish to construct a data structure (called an oracle) for a directed 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).
- The path queries corresponding to the above distance queries

There are two main motivations for this problem. The first is modeling a network where vertices (or edges) occasionally fail. We might not be able to afford to stall distance queries and recompute shortest paths every time a vertex fails. We could use a dynamic shortest path algorithm to reduce the stall time, but this would still require lengthy update periods. A sensitivity oracle, on the other hand, allows us to prepare for a single failure ahead of time. That is, when a vertex or edge fails, we can continue to answer queries quickly with our oracle while constructing a new oracle (for the graph with a vertex deleted) in the background. Our construction time is only $\widetilde{O}$(mn) [1], so as long as failures are relatively rare, we can finish constructing the new oracle before another vertex fails.

The second motivation is Vickrey pricing [10]. This is a method for determining the value of an edge in a network where edges are owned by independent agents. If we want to send information between two points, then intuitively, the value of an edge e depends on how much harder it would be to send that information without using e. In particular, we determine the value of e by comparing the original shortest path between the points to the shortest path that avoids e. This is precisely what distance sensitivity oracles allow us to compute, so they are by far the fastest option if we want to find Vickrey prices for many shortest paths in a graph.

1.2 Existing Algorithms

The naive approach to our problem would be to store the shortest distance between every pair (x, y) avoiding every edge (u, v), but this would require $O(mn^2)$ space and m all pairs shortest path computations. We can improve upon this by noticing that for any pair (x, y), removing an edge that was not on the original shortest path does not change anything. Thus, for the pair (x,y), we only have to store distances avoiding the $O(n)$ edges on the original path. Moreover, only edges in the shortest path tree of x affect distances from x, so we can compute all the necessary information by

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

doing n single source shortest path computations per vertex (one for each edge in the shortest path tree). Using Dijkstra's algorithm [7] to compute shortest paths, this technique reduces the space to $O(n^3)$ and the construction time to $\widetilde{O}(mn^2)$. The query is just a table look up so it takes constant time.

The first non-trivial algorithm for the problem was developed by Demetrescu *et al* [6]. They managed to keep the constant query time and the $\widetilde{O}(mn^2)$ construction time while reducing the space requirement to $\widetilde{O}(n^2)$. Soon afterwards, Chowdhury and Ramach [5] claimed to have developed an oracle with a construction time of $\widetilde{O}(mn)$, but they later discovered a mistake in their algorithm. The state of the art algorithm was developed by Bernstein and Karger [4]. Their oracle also had a constant query time and an $\widetilde{O}(n^2)$ space requirement, but the construction time was improved to $\widetilde{O}(n^2\sqrt{m})$. However, unlike in the other papers, the construction was randomized (Monte Carlo).

1.3 Our Contributions

We present an oracle with $O(1)$ query time, $O(n^2 \log n)$ space requirement, and a $O(mn \log n + n^2 \log^2 n) = \widetilde{O}(mn)$ randomized (Monte Carlo) construction time. We also present a deterministic oracle, with space and time bounds worse by a factor of $O(\log n)$. This is a surprising result because $O(1)$ query, $O(mn)$ construction time, and $O(n^2)$ space is the best known bound (up to log factors) for the simpler problem APSP: finding all pairs shortest paths in a weighted, directed graph. Thus, barring an improvement for APSP, our oracle is optimal up to log factors.

Our oracle is based on two novel techniques. Bernstein and Karger [4] showed that the problem of having to avoid every vertex on a particular x-y path can be reduced to that of avoiding just a few sub-paths of the x-y path. But avoiding whole sub-paths is hard, so in section 4 we show that we can actually get away with just avoiding a single "key" vertex on each sub-path. In particular, the key vertex of a sub-path is the single vertex on that sub-path that is hardest to avoid. Thus, we have reduced the problem of avoiding every vertex on an x-y path to that of only avoiding a few key vertices.

Section 5 assumes that we have already found these key vertices, and it presents an algorithm for actually avoiding them. The basic idea is that we follow the intuition behind Dijkstra's single source shortest path algorithm [7] and express the shortest distance to x avoiding a key vertex as a function of the distances to the neighbors of x. The resulting recurrence relation is somewhat more complicated than that of Dijkstra's algorithm, but we show that we can nonetheless turn our recurrence relation into an instance of a shortest path problem. That is, we create a new graph (with a source) where each value we want to compute (a shortest distance avoiding a key vertex) is the shortest distance to some vertex in the new graph. We then compute our values by running Dijkstra's algorithm on this graph.

More generally, section 5 presents a somewhat different way of thinking about dynamic programming. Given a set of values to compute, our approach does not require us to explicitly define the order in which we compute these values. Instead, we simply express each value as a function of the other values. This leads to a natural dynamic programming graph where vertices correspond to values we want to compute, and edges correspond to dependencies between these values. By running Dijkstra's algorithm on this graph, we are able to implicitly determine the correct order of computation.

Finally, section 6 presents an efficient algorithm for finding the key vertices on every shortest path.

Remark 1. We only show how to answer distance queries avoiding a failed vertex because it is easy to extend our oracle to answer the other queries, without increasing the space or time parameters. See Demetrescu *et al* [6] for a description of how an oracle for avoiding vertices can also be used to avoid edges. Path queries take O(L) time to answer, where L is the length of the output path (the same query time as for the traditional all pairs shortest paths data structure).

1.4 Related Work

As mentioned in section 1.1, our oracle allows us to find the edge whose loss is most damaging to a given shortest path. That is, the edge whose removal causes the maximum increase in the shortest distance between the two points. A natural generalization would be to find the k edges whose removal causes the largest change in distance. However, Bar-Noy *et al* [2] showed that this problem is NP-hard for general k.

The single-pair version of the distance sensitivity problem is known as the replacement path problem. Given a pair of vertices (s, t) we must find, for every edge on the shortest path between s and t, the new shortest path avoiding that edge. The naive approach would be to remove each edge, one at a time, and compute shortest paths from s each time. This takes $\widetilde{O}$(mn) time. In fact, no $o(mn)$ approach is known for the general case of weighted, directed graphs. This is surprising because our sensitivity oracle implicitly computes replacement paths for *all pairs*, yet it has the same $\widetilde{O}(mn)$ construction time (although it is slower by a polylogarthimic factor). The only known lower bound for replacement paths in general graphs, proved by Hershberger *et al* [11], is $O(m\sqrt{n})$ in the path comparison model.

There exist much faster algorithms for the replacement path problem in specific classes of graphs. In *undirected* graphs, Ball *et al* [1] presented an algorithm with an $O(m + n \log n)$ running time. For directed, *unweighted* graphs, Roditty and Zwick [13] developed an $\widetilde{O}(m\sqrt{n})$ algorithm. Finally, Emek *et al* [8] developed an $\widetilde{O}(n)$ algorithm for general *planar* graphs.

2. NOTATION

We use the same notation as Demetrescu *et al* [6]. Let G = (V,E) be the graph in question. We denote the edge from u to v (if it exists) by (u,v), and we let w(u,v) be its weight. Like other papers in this field, we assume W.L.O.G that shortest paths are unique, since we can always add small perturbations to break any ties.

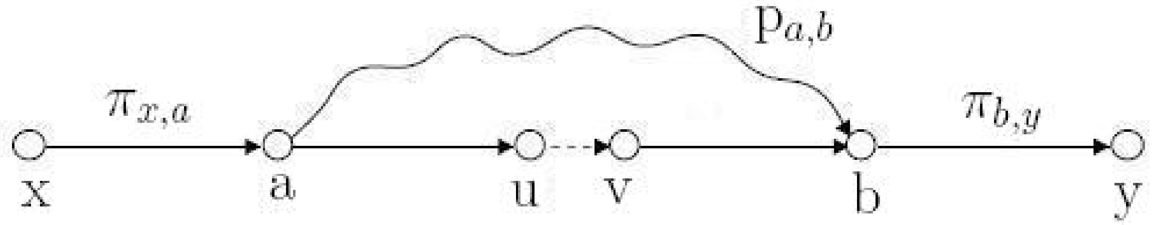

Figure 1: A detour avoiding the interval [u,v] on $\pi_{x,y}$

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, and let $\widehat{\pi}_{x,y}$ be the shortest x-y path in $\widehat{G}$. 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 $w(\pi)$ be the weight of a path π, and let $d_{x,y} = w(\pi_{x,y})$.

Also, let $\pi_{x,y,S}$ be the shortest path from x to y that avoids the set of nodes S, and define $d_{x,y,S}$ analogously. For simplicity, we write $\pi_{x,y,\{v\}}$ as $\pi_{x,y,v}$.

Finally, let T_x be the shortest path tree rooted at vertex x, and define $\widehat{T_x}$ analogously for $\widehat{G}$. Given any vertex v, Let $T_x(v)$ be the subtree of T_x that is rooted at v.

3. AN OVERVIEW OF EXISTING TECHNIQUES

Our algorithm relies on machinery developed by Demetrescu *et al* [6] and Bernstein and Karger [4], so we start with an overview of existing techniques. The first step is compute and store all-pairs shortest paths (no failed vertices), which can be done in $\widetilde{O}(mn)$ time using Dijkstra's algorithm [7]

3.1 Detours

Definition 1. Let a,b be vertices on $\pi_{x,y}$. If x and y are clear from context we say a < b if a comes before b on $\pi_{x,y}$. Assuming a ≤ b, let the interval [a,b] be the set of vertices v such that a ≤ v ≤ b.

We now present a basic property of paths avoiding failed vertices. Let $u < v$ be vertices on $\pi_{x,y}$ and say that we want to find $\pi_{x,y,[u,v]}$. Since $\pi_{x,y,[u,v]}$ avoids $[u,v]$, it needs to deviate from $\pi_{x,y}$ at some vertex $a < u$, and then merge back at $b > v$. Moreover, $\pi_{x,y,[u,v]}$ cannot 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 only deviate from a'. Similarly, $\pi_{x,y,[u,v]}$ only merges back at one vertex $b > v$ (Figure 1). This yields:

Definition 2. Let x ≤ a ≤ b ≤ y be vertices on $\pi_{x,y}$. A path $p_{a,b}$ from a to b is said to be a *detour* of $\pi_{x,y}$ if $p_{a,b} \bigcap \pi_{x,y} = \{a,b\}$

LEMMA 3.1. *[6] 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 path concatenation, and $p_{a,b}$ is a detour of $\pi_{x,y}$ such that $p_{a,b} = \pi_{a,b,[u,v]}$ (Figure 1).*

3.2 Path Cover

One difficulty we have to overcome is that it takes $O(n^3)$ space to naively store $d_{x,y,v}$ for all triplets (x,y,v). The solution is to decompose every path $\pi_{x,y}$ into a small number of intervals, and store $d_{x,y,I}$ for each interval I (instead of storing every $d_{x,y,v}$). We now present a lemma of Demetrescu *et al* [6] which shows how $d_{x,y,I}$ relates to $d_{x,y,v}$ for $v \in I$.

LEMMA 3.2. **The Path Cover Lemma** *[6]*
Let $x \le s < v < t \le y$ be vertices on $\pi_{x,y}$ (v is the 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. (sketch) By Lemma 3.1, we get that each term in the min clause corresponds to one of three cases: $\pi_{x,y,v}$ diverges from $\pi_{x,y}$ after s, merges before t, or avoids all of $[s,t]$. □

3.3 Covering With Centers

The path cover lemma suggests an approach for only storing a small number of values. To compute $d_{x,y,v}$, we simply need to find vertices s and t to the left and right of v for which we already store $d_{s,y,v}$, $d_{x,t,v}$, and $d_{x,y,[s,t]}$. Just as Bernstein and Karger [4], we do this by designating some vertices as *centers*, which store more information than ordinary vertices.

Definition 3. Given $x, y \in V$, we say that x *covers* v in our oracle if we store $d_{x,y,v}$ for every y in $T_x(v)$ (defined in notation section). That is, x covers v if we store all shortest distances from x avoiding v.

By the intuition above, we want to find centers $c_x \in \pi_{x,v}, c_y \in \pi_{v,y}$ that cover v; this will give us the first two terms of the path cover lemma, which is a good start (of course, c_y should cover v in $\widehat{G}$ because we want the distance *to* c_y). We could ensure that c_x and c_y cover v by making centers cover every vertex, but then we could only afford a small number of centers because covering takes space per vertex. Thus, we use a slightly different approach.

Say, for intuition, that we picked our centers randomly from V. Then, a path with many vertices is likely to contain a

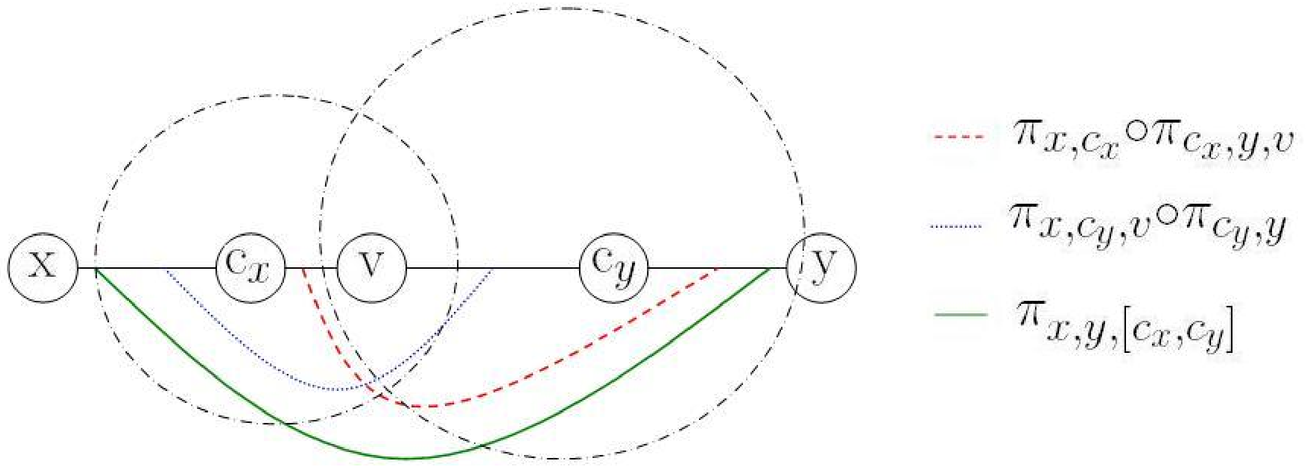
X
c_x
V
c_y
y
$\pi_{x,c_x} \circ \pi_{c_x,y,v}$
$\pi_{x,c_y,v} \circ \pi_{c_y,y}$
$\pi_{x,y,[c_x,c_y]}$

Figure 2: Using centers to apply the path cover lemma

center. But if $\pi_{x,v}$ has few vertices then we would need to sample a lot of centers to ensure a center on $\pi_{x,v}$. Note, however, that if $\pi_{x,v}$ has few vertices then a center c_x on $\pi_{x,v}$ would still cover v even if c_x only covered vertices in a small ball around itself, as opposed to covering *every* vertex (Figure 2). So we have different types of centers: rare ones cover large balls, while common ones cover small balls.

But what about the third term in the path cover lemma? By the argument above, this term will have the form $d_{x,y,[c_x,c_y]}$, where c_x and c_y are centers that cover v. More formally:

Definition 4. A *covering chain* of $\pi_{x,y}$ is a sequence of centers c$_1$, ... ,c$_j$ such that [c$_1$, c$_j$] = $\pi_{x,y}$, and c$_i$ covers all vertices in $[c_{i-1}, c_i]$, and $[c_i, c_{i+1}]$ (see figure 3). That is, it is a decomposition of $\pi_{x,y}$ into intervals such that all the vertices in an interval are covered by the endpoints of that interval. We refer to the intervals $[c_i, c_{i+1}]$ as *covering intervals* of $\pi_{x,y}$

Note that if we have a covering chain for $\pi_{x,y}$ and we store d$_{x,y,I}$ for all covering intervals I = $[c_i, c_{i+1}]$, then we can efficiently compute d$_{x,y,v}$ for any v. For given v $\in \pi_{x,y}$, let $[c_i, c_{i+1}]$ be the interval containing v; we know that c_i and c_{i+1} cover v, so if we also know $d_{x,y,[c_i,c_{i+1}]}$ we can use the path cover lemma. Of course, we can only afford to store $d_{x,y,I}$ for a small number of intervals I, so section 3.5 describes how we can pick centers that ensure the existence of covering chains with few intervals.

3.4 Excluding Vertices

In order for this approach to have a small preprocessing time, we need an efficient algorithm for making a common center cover a small ball around itself. The problem is that we cannot just naively avoid every vertex in the ball because a "small" ball may still contain many vertices: "small"' only refers to the number of edges in shortest paths within the ball.

Given some source x, and a failed node v, we can trivially compute d$_{x,y,v}$ for all y $\in$ V in $\widetilde{O}$(m) time by doing a single-source shortest path computation on the graph G - {v}. But this is wasteful because removing a vertex might only affect small portions of T$_x$ (recall that T_x is the shortest path tree from x), in which case we would like to avoid examining all of G.

Demetrescu *et al* [6] formalize this idea. Let x be our source, and let v be the vertex we want to avoid. Recall that $T_x(v)$ is defined as the subtree of T_x rooted at v. Note that removing v only affects vertices in $T_x(v)$ because if $y \notin T_x(v)$, then we must have $d_{x,y,v} = d_{x,y}$. Thus, intuitively, we only need to focus on vertices in T$_x(v)$ (and edges incident upon those vertices). More formally:

LEMMA 3.3. *[6] Given a source x and a vertex v, we can make x cover v in O($|T_x(v)|$) time (here, $|T_x(v)|$ also encompasses edges incident upon vertices in $T_x(v)$).*

Definition 5. Let $L_x(L)$ be the set of all vertices at level L in T_x. That is $L_x(L)$ contains all v for which $\pi_{x,v}$ contains exactly L edges.

COROLLARY 3.4. *[6] Given a source x and a level L, we can make x cover all vertices in $L_x(L)$ in $\widetilde{O}(m)$ time. That is, we can compute $d_{x,y,v}$ $\forall$ $y \in$ V, $v \in L_x(L)$.*

Remark 2. This corollary implies that we can make x cover all vertices at level $\leq$ L in $\widetilde{O}$(mL) time. This is exactly what we wanted since a "small ball" around x is precisely the set of vertices of small level in T_x.

PROOF. All vertices at level L in T$_x$ have disjoint subtrees, so by lemma 3.3 we explore each vertex in the graph at most once. Similarly, each directed edge is incident upon at most one subtree. □

3.5 Picking Centers

We use the method of Bernstein and Karger [4] to pick centers that ensure the existence of covering chains with few intervals. To formalize the idea of large and small centers we use $O(\log n)$ *priorities*: centers with low priority are common, but only cover small balls.

Definition 6. 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

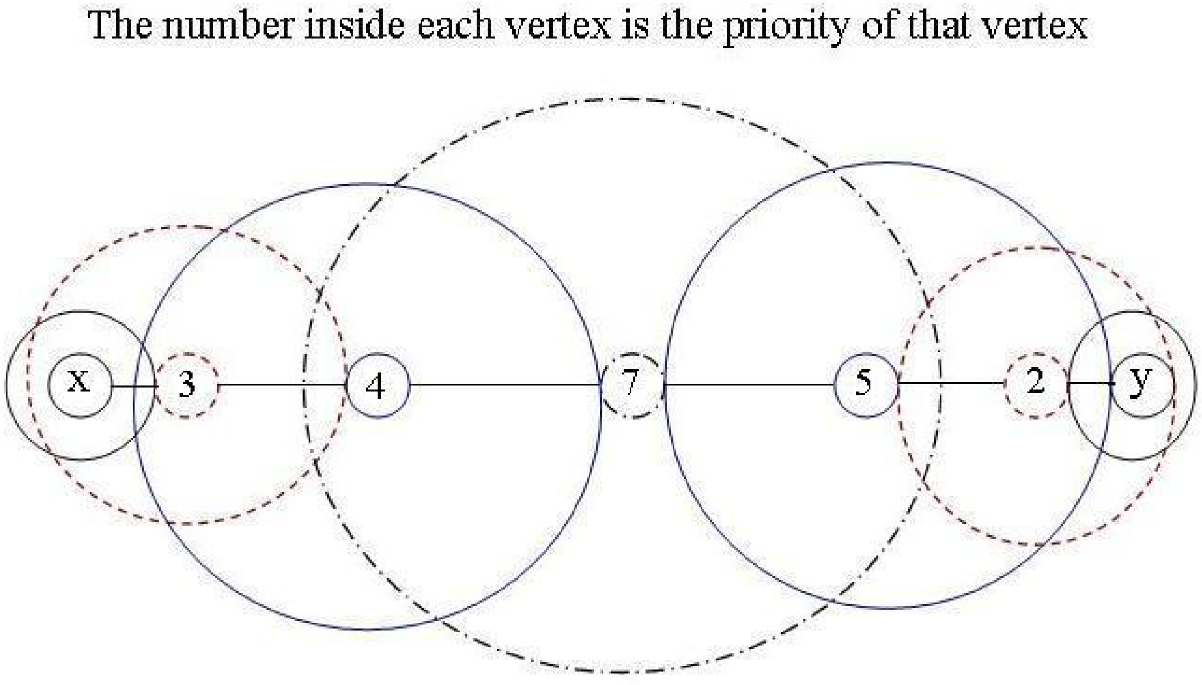
The number inside each vertex is the priority of that vertex
x
3
4
7
5
2
y

Figure 3: An example of a covering chain for $\pi_{x,y}$. Note that every vertex is a 1-center, so even though x, y are arbitrary, they still cover tiny balls

Desired Properties: We require that $|R_k| = \widetilde{O}(n/2^k)$, and that any shortest path with $\widetilde{O}(2^k)$ vertices contains a k-center. Again, note that high priority centers are rare.

Picking Centers: An easy approach is to obtain R_k by sampling each vertex, independently, with probability $\Theta(1/2^k)$. This ensures that the desired properties hold with high probability (see Bernstein and Karger [4]). See Remark 4.1 for a deterministic construction.

Center Information: We make a k-center c cover all vertices in T_c (and $\widehat{T_c}$) that are not in the subtree of some (k+1)-center. That is, c moves down T_c, covering vertices until it reaches a (k+1)-center. Note that for any x, y, v, the biggest center on $\pi_{x,v}$ covers v, as does the biggest center on $\pi_{v,y}$ (the latter covers v in $\widehat{G}$).

Covering Chains: Our choice of centers leads to a very natural covering chain with few intervals. Given a path $\pi_{x,y}$, we find a list of centers in ascending priority; so $c_1 = x$, c_2 is the first center on $\pi_{x,y}$ bigger than c_1, c_3 is the first center bigger than c_2, and so on. Note that there can only be $O(\log n)$ centers in this list because there are $O(\log n)$ center priorities. Once we get to the biggest center on $\pi_{x,y}$ we begin to descend in priority in a similar fashion (see Figure 3). It is easy to verify that this is indeed a covering chain.

We store this information in a lookup table D_k, where D_k[c,y,v] stores $d_{c,y,v}$ if c is a center that covers v. We also store $\widehat{D}_k$ for $\widehat{G}$. As proved in section 5 of Bernstein and Karger [4], we get:

THEOREM 3.5. *[4] We can initialize D_k in $\widetilde{O}(mn)$ time, and $\widetilde{O}(n^2)$ space.*

PROOF. (sketch) Since any shortest path with $\widetilde{O}(2^k)$ vertices contains a (k+1)-center, any k-center c only has to cover vertices up to depth $\widetilde{O}(2^k)$ in T_c. Thus, each k-center only requires $\widetilde{O}(n{\cdot}2^k)$ space, and by lemma 3.3, all $\widetilde{O}(2^k)$ levels can be covered in $\widetilde{O}(m{\cdot}2^k)$ time. The theorem then follows from the fact that there are $\widetilde{O}(n/2^k)$ k-centers. □

4. BOTTLENECK VERTICES

Recall that our choice of centers leads to a covering chain with few intervals for each shortest path $\pi_{x,y}$. In particular, given any $v \in \pi_{x,y}$ the endpoints of the interval that contains v are guaranteed to cover v, so we can compute the first two terms of the path cover lemma. Thus, all we have left to do is compute $d_{x,y,I}$ for each interval I on the chain (this is the last term of the path cover lemma).

Unfortunately, we do not know how to compute $d_{x,y,I}$ efficiently. Bernstein and Karger [4] overcome this problem by showing that instead of storing $d_{x,y,I}$, we can store any function $F_{x,y,I}$ that satisfies certain requirements. But even this is difficult because it requires us to work with whole intervals. Our solution is to show that each interval can be effectively condensed to a "key" vertex, so that instead of having to avoid the whole interval, we can just avoid that one vertex.

Note that every interval I on $\pi_{x,y}$ contains some vertex w that is hardest to avoid (w maximizes $d_{x,y,w}$ over $w \in I$).

We show that instead of avoiding all of I, we can get away with avoiding w.

Definition 7. Given an interval I on $\pi_{x,y}$, define the *bottleneck* vertex of I (with respect to x and y) to be w = $\text{argmax}_{v \in I}\{d_{x,y,v}\}$. We sometimes refer to w simply as the bottleneck of I.

LEMMA 4.1. **Bottleneck Lemma** *[4] Let $x \leq s < v < t \leq y$ be vertices on $\pi_{x,y}$ (v is the failed vertex), and let w be the bottleneck of [s,t]. Then,*

$$d_{x,y,v} = min\{d_{x,s} + d_{s,y,v}, d_{x,t,v} + d_{t,y}, d_{x,y,w}\}$$

Note that this is almost identical to the path cover lemma, except that instead of avoiding all of [s,t], we just avoid the bottleneck of [s,t].

PROOF. We consider two cases. If $\pi_{x,y,v}$ avoids all of $[s,t]$ then it is a feasible path avoiding w (the bottleneck), so $d_{x,y,v} \geq d_{x,y,w}$; but w is the bottleneck of [s,t], so we cannot have $d_{x,y,v} > d_{x,y,w}$, so we must have $d_{x,y,v} = d_{x,y,w}$. If $\pi_{x,y,v}$ does not avoid all of [s,t] then it must go through either s or t, in which case $d_{x,y,v} = d_{x,s} + d_{s,y,v}$ or $d_{x,t,v} + d_{t,y}$. □

Thus, instead of avoiding every interval on a covering chain, we can avoid the bottleneck vertex of each interval, which is substantially easier. We are now ready to describe our general framework.

Definition 8. Now that we have constructed a covering chain for every path $\pi_{x,y}$ (section 3.5), we define $CI[x,y,i]$ to be the ith covering interval on the covering chain for $\pi_{x,y}$ (see definition 4 for covering interval). Recall that every $v \in \pi_{x,y}$ is contained by some covering interval $CI[x,y,i]$.

Definition 9. Define BV[x,y,i] to be the bottleneck vertex of CI[x,y,i].

Definition 10. Given v on $\pi_{x,y}$, and letting $CI[x,y,i] = [c_x, c_y]$ be the covering interval on $\pi_{x,y}$ that contains v, we define

$$MTC(x,y,v) = \min\{d_{x,c_x} + d_{c_x,y,v}, d_{x,c_y,v} + d_{c_y,y}\}$$

(that is, the shortest path through the two centers covering v). MTC stands for minimum through centers. We refer to the path corresponding to MTC(x,y,v) as the *shortest centered path* from x to y avoiding v. Note that by the bottleneck lemma,

$$d_{x,y,v} = \min\{MTC(x,y,v), d_{x,y,BV[x,y,i]}\}$$

THEOREM 4.2. *Once we compute $d_{x,y,BV[x,y,i]}$ for every triplet (x,y,i) (i is a center priority), we can construct an $\widetilde{O}(n^2)$ space oracle that can compute $d_{x,y,v}$ in constant time for any triplet (x,y,v).*

PROOF. Our proof directly follows the one used in section 6 of Bernstein and Karger [4], except that bottleneck vertices allow us to avoid dealing with whole intervals, so where they use $EP[x,y,i]$ we just use $d_{x,y,BV[x,y,i]}$. Except for a few auxiliary structures (which we omit – see Bernstein and Karger [4] for details), the only things we store are the bottleneck values $d_{x,y,BV[x,y,i]}$, the center information table D_k, and the shortest distances in the original graph (no failed vertices).

Given a triplet (x,y,v), we find the endpoints of the covering interval $CI[x,y,i]$ that contains v (this is easy to do). Because of how we constructed covering chains, these endpoints must cover v, so using our center information we can find the shortest paths that go through these endpoints. This gives us the first two terms of the bottleneck lemma. We know the third term because we have avoided the bottleneck vertex on every covering interval. Thus, we can find $d_{x,y,v}$ in constant time by simply taking the minimum of these three terms. The space is $\widetilde{O}(n^2)$ because D_k (the center information table) requires $\widetilde{O}(n^2)$ space (Theorem 3.5), as does storing every $d_{x,y,BV[x,y,i]}$: there are only $O(\log n)$ covering intervals per path $\pi_{x,y}$, so there are $\widetilde{O}(n^2)$ bottlenecks in total (one per covering interval). □

THEOREM 4.3. *If we can compute $d_{x,y,BV[x,y,i]}$ for every triplet (x,y,i) in a total of T time, then we can construct the framework of Theorem 4.2 in $O(T) + \widetilde{O}(mn)$ time.*

PROOF. Once we have the bottleneck values $d_{x,y,BV[x,y,i]}$, all we have left is to construct the center information table D_k and some auxiliary structures. By theorem 3.5 we can construct D_k in $\widetilde{O}$(mn) time; we can construct all of the auxiliary structures even faster. See Section 6 of Bernstein and Karger [4] for details on the auxiliary structures. □

Remark 1. Bernstein and Karger [4] gave a randomized construction of D_k, but there exists a deterministic construction. We can directly apply a technique of King [12] to deterministically pick centers with the required properties specified in section 3.3. This construction is less efficient in both time and space by a factor of $O(\log(n))$.

5. AVOIDING BOTTLENECK VERTICES

There are two things left to show: how to find a bottleneck BV[x,y,i] for each CI[x,y,i] (definitions 8, 9), and how to compute $d_{x,y,BV[x,y,i]}$ once we know BV[x,y,i]. In this section, we focus on the latter, so we assume the following theorem.

THEOREM 5.1. *We can find BV[x,y,i] for every triplet (x,y,i) in a total of $\widetilde{O}(n^2)$ time.*

PROOF. Section 6 describes an algorithm for doing this. Intuitively, the bottleneck for CI[x,y,i] is the vertex that maximizes a certain function, which we show to be rather well behaved. This allows us to binary search on CI[x,y,i] to find the bottleneck vertex. □

5.1 A Recurrence Relation for Bottleneck Values

To recap, theorem 4.2 reduces the problem of computing O(n^3) different values $d_{x,y,v}$ to the simpler problem of computing $\widetilde{O}(n^2)$ different bottleneck values $d_{x,y,BV[x,y,i]}$. To solve this simpler problem, we rely on the intuition behind Dijkstra's single source shortest path algorithm [7]: we express the distance to a vertex as a function of the distances to its neighbors.

More formally, for any vertex y, let $IN(y) = \{y' \in V \mid (y', y) \in E\}$. We know that for any triplet (x,y,v) we have

$$d_{x,y,v} = \min_{y' \in IN(y)} (d_{x,y',v} + w(y', y))$$

But this is a recurrence relation between all $O(n^3)$ values $d_{x,y,v}$, and we cannot afford to look at $O(n^3)$ values. Instead, we want a recurrence relation between the $\widetilde{O}(n^2)$ bottleneck values.

So what is our relation for $d_{x,y,BV[x,y,i]}$? Well, letting $v = BV[x, y, i]$, we start in the same fashion: we recall that $d_{x,y,v} = \min_{y' \in IN(y)} (d_{x,y',v} + w(y', y))$. But since $d_{x,y',v}$ itself might not be a bottleneck value, we use the bottleneck lemma to express it as the minimum of a bottleneck value and an MTC term. This gives us:

$$\begin{aligned} d_{x,y,v} &= \min_{y' \in IN(y)} (d_{x,y',v} + w(y', y)) \\ &= \min_{y' \in IN(y)} (\min\{MTC(x, y', v), d_{x,y',BV[x,y',j]}\} + w(y', y)) \end{aligned}$$

where j in $BV[x, y', j]$ is the center priority for which $CI[x, y', j]$ contains v. Rearranging, we get

$$\begin{aligned} d_{x,y,v} = \min\{ &\min_{y' \in IN(y)} (MTC(x, y', v) + w(y', y))(\text{term 1}), \\ &\min_{y' \in IN(y)} (d_{x,y',BV[x,y',j]} + w(y', y))(\text{term 2})\} \end{aligned} \quad (1)$$

(technical note: there may be some $y' \in IN(y)$ for which v is not on $\pi_{x,y'}$, so $MTC(x, y', v)$ and $BV[x, y', j]$ are not even defined. But in this case we simply have $d_{x,y',v} = d_{x,y'}$, which we already know. Thus, we can handle this special case by just defining $MTC(x, y', v)$ to be $d_{x,y'}$ (when $v \notin \pi_{x,y'}$), and defining $d_{x,y',BV[x,y',j]}$ to be infinity).

At first, this may not seem like a relation between the bottleneck values because we have all these MTC terms. But recall that we can compute any $MTC(x, y, v)$ in constant time using the center information that we precomputed in section 3.5. Thus, the MTC terms are just constants, so Equation (1) is in fact a direct recurrence relation (recall that in the equation $v = BV[x, y, i]$, so $d_{x,y,v}$ is also a bottleneck value).

5.2 Using the Recurrence Relation

Notice that the recurrence in Equation (1) closely resembles the recurrence in Dijkstra's algorithm, and would resemble it even more if not for term 1 in the equation. The main difference is that term 1 sets an initial constant upper bound on each bottleneck value. This suggests the possibility of using Dijkstra's algorithm to solve our recurrence.

Let us make this more explicit. We create a new directed graph $G_{bv} = (V_{bv}, E_{bv})$ with non-negative weight function w_{bv}. We let V_{bv} consist of a source s and the vertices $\{v[x, y, i]\}$: our goal is to construct the graph in such a way that the shortest distance from s to $v[x, y, i]$ is precisely $d_{x,y,BV[x,y,i]}$. To construct E_{bv} note that since term 2 in Equation (1) is precisely the recurrence relation in Dijkstra's algorithm, we just include the edges that are implicitly present in that second term. In particular, we add an edge from $v[x, y', j]$ to $v[x, y, i]$ if $y' \in IN(y)$ and j is the index for which $CI[x, y', j]$ contains $BV[x, y, i]$. We set the weight of edge $(v[x, y', j], v[x, y, i])$ to be $w(y', y)$.

But what about the first term in Equation (1)? Well, having this initial upper bound is equivalent to adding an edge from the source s to every vertex $v[x, y, i]$ with weight equal to term 1 in Equation (1) (this term is different for each $d_{x,y,BV[x,y,i]}$). The reason for this is that the first step of Dijkstra's algorithm relaxes all edges from the source, which effectively initializes each vertex $v[x, y, i]$ with the constant in term 1. Dijkstra's algorithm then tries to find shorter distances by exploiting the dependencies between the vertices (*i.e.* it tries to find shortest paths that do not directly use the source edge).

Since the base case and the recurrence for shortest paths in G_{bv} is the same as our recurrence for detour paths in Equation (1), we indeed have that the shortest distance from s to $v[x, y, i]$ is $d_{x,y,BV[x,y,i]}$. Thus, we can compute the bottleneck values in $\widetilde{O}(|V_{bv}| + |E_{bv}|)$ time by just running Dijkstra's algorithm on G_{bv} with source s. But note that for every $v[x, y, i] \in V_{bv}$ we added at most $|IN(y)|$ edges to E_{bv}. Since every y is in $O(n \log(n))$ triplets (x, y, i) we have $|E_{bv}| = O(n \log(n) \sum_{y \in V} |IN(y)|) = O(mn \log(n))$, so the total running time is $\widetilde{O}(mn)$.

Remark 2. Note that in general terms, our algorithm can be thought of as a different way to go about dynamic programming. If we are given a set of values that we want to compute, traditional dynamic programming requires us to explicitly define an order in which to compute these values. For example, we might know to put our values into a table $M[m, n]$ and then compute in the order $M[1, 1], M[2, 1], M[1, 2], M[3, 1], M[2, 2], ..., M[m, n]$.

Our approach, on the other hand, does not require us to determine this order ahead of time. Instead, all we do is express each value as a function of the other values. In particular, we store our values in a directed graph instead of a table, where each vertex corresponds to a value we want to compute, and the value at a vertex is a function of the values of its neighbors. As long as this function satisfies certain "monotonicity" properties, Dijkstra's algorithm will implicitly discover the correct order of computation for us. By "monotonicity" properties, we simply mean that if we need the value at u to compute the value at v then the value at u must be smaller than the value at v (e.g. the triangle inequality for shortest paths).

This approach bears some similarity to memoization, but it is more powerful because the use of a Dijkstra like algorithm on our dependency graph allows us to implicitly break cycles for a large class of dependency functions. For example, our approach encompasses shortest paths in general graphs, while memoization only encompasses shortest paths in acyclic graphs.

6. FINDING BOTTLENECK VERTICES

We now turn to proving theorem 5.1 in section 5 (see section 4 for a review of basic definitions). We focus on efficiently finding the bottleneck of a specific interval $CI[x, y, i] = [c_x, c_y]$. The following lemma provides a useful way to think about bottleneck vertices. (technical note: a covering interval can have multiple bottleneck vertices, but it does not matter which one we choose, so for the sake of clarity we assume that the bottleneck of $CI[x, y, i]$ is unique).

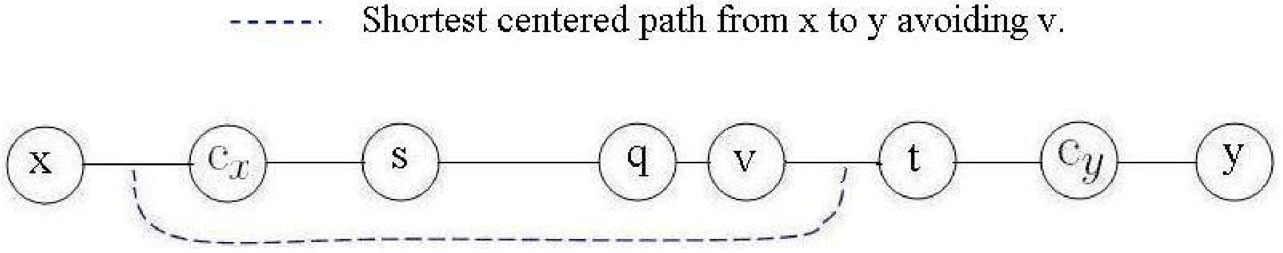

Figure 4: The case in algorithm FindBot([s,t]) where $L(x, y, v) > R(x, y, v)$

LEMMA 6.1. *The bottleneck of CI[x,y,i] is the vertex $w \in CI[x, y, i]$ that maximizes $MTC(x, y, w)$. In other words, removing the bottleneck vertex maximizes the distance through the centers.*

PROOF. By definition, the bottleneck vertex maximizes $d_{x,y,w}$ among $w \in CI[x, y, i]$. But by the path cover lemma, $d_{x,y,w} =$
$\min\{MTC(x, y, w), d_{x,y,CI[x,y,i]}\}$. The second term is the same for all vertices in $CI[x, y, i]$, so we only have to worry about maximizing the first term. □

Definition 11. Given v in $[c_x, c_y]$ define

$$L(x, y, v) = d_{x,c_x} + d_{c_x,y,v}$$

That is, $L(x, y, v)$ is the length of the shortest path avoiding v that goes through c_x. Similarly, let

$$R(x, y, v) = d_{x,c_y,v} + d_{c_y,y}$$

Then

$$MTC(x, y, v) = \min\{L(x, y, v), R(x, y, v)\}$$

Thus, our goal is to find the vertex that maximizes the minimum of $\{L(x, y, v), R(x, y, v)\}$. Note that using our precomputed center information from section 3.5, we can compute $R(x, y, v)$ and $L(x, y, v)$ in constant time for any vertex v. The problem is that individually checking every vertex in $[c_x, c_y]$ takes too long.

Thus, the basic intuition behind out algorithm is that instead of computing $L(x, y, v), R(x, y, v)$ for *every* vertex, we will only compute it for a few specific vertices. For each such vertex v, comparing $L(x, y, v)$ and $R(x, y, v)$ will gives us information about the shortest *centered* path avoiding v (see definition 10), which will in turn give us information about where to look for the bottleneck. The following lemma elucidates how $L(x, y, v)$ and $R(x, y, v)$ relate to the shortest centered path.

LEMMA 6.2. *Let v be some vertex in $[c_x, c_y]$, and let π be the shortest* centered *path avoiding v.*
• *if $L(x, y, v) < R(x, y, v)$ then $MTC(x, y, v) = L(x, y, v)$ and π goes through c_x but not c_y.*
• *if $L(x, y, v) > R(x, y, v)$ then $MTC(x, y, v) = R(x, y, v)$ and π goes through c_y but not c_x.*
• *if $L(x, y, v) = R(x, y, v)$ then $MTC(x, y, v) = L(x, y, v) = R(x, y, v)$*

PROOF. This stems directly from the definitions, since $L(x, y, v)$ is the shortest distance through c_x, $R(x, y, v)$ is the shortest distance through c_y, and the shortest *centered* path must go through at least one of c_x or c_y. □

Lemma 6.2 only gives us information about the shortest centered path avoiding a specific v, so the question is which vertices will give us information about the interval as a whole. We want to pick vertices that already have some significance, so we will pick vertices with large values for $L(x, y, v)$. Note that maximizing just $L(x, y, v)$ is much easier than maximizing $\min\{L(x, y, v), R(x, y, v)\}$

Definition 1. Given an interval $I \subset [c_x, c_y]$ let $v_L(x, y, I)$ be the vertex v in I that maximizes $L(x, y, v)$. That is, $v_L(x, y, I) = \text{argmax}_{v \in I} L(x, y, v)$

So intuitively, we want to use v_L to find important vertices v, and then use lemma 6.2 to glean information about these vertices that tells us where to look next. We first show how to compute $v_L(x, y, I)$ in constant time, and then use this to compute a bottleneck for $[c_x, c_y]$ in $O(\log(n))$ time.

By definition 1, to compute $v_L(x, y, I)$ we just need to compute $\text{argmax}_{v \in I}(d_{c_x,y,v})$. But we already know $d_{c_x,y,v}$ for every $v \in [c_x, c_y] \supset I$ (this is our center information). The naive way of storing this center information is to use an array A where $A[i] = d_{c_x,y,v_i}$ (v_i is the ith vertex on $[c_x, c_y]$). To compute v_L we need a more complicated structure that allows us to find the maximum value in any sub-array I of A in constant time.

Fortunately, this structure already exists: it is called the range maximum query data structure [3]. An array can be turned into a range maximum data structure in linear space and time, so using range maximum data structures does not increase the required space or preprocessing time of our center information. note that although we cannot afford to spend linear time for finding every bottleneck vertex, we can afford linear time in our center information. The reason for this is that our center information is indexed by a center and a vertex covered by that center, whereas bottleneck vertices are indexed by two arbitrary vertices.

We now present a recursive algorithm FindBot, where given any $I = [s, t] \subset [c_x, c_y]$, FindBot(I) returns the bottleneck vertex of I (that is the vertex $v \in I$ that maximizes $MTC(x, y, v)$) – see figure 5 for pseudo code. In essence, FindBot is just a binary search. Let q be the midpoint of $[s, t]$, and let $v = v_L(x, y, [q, t])$. We now consider two cases (for the sake of clarity we ignore minor technical details such as the case where $[s, t]$ only contains 1 vertex. These are handled in the pseudo code).

Suppose that $L(x,y,v) \leq R(x,y,v)$. Then, by definition, $MTC(x,y,v) = L(x,y,v)$. But for any other $w \in [q,t]$ we have

$$\begin{aligned} & MTC(x,y,w) \\ & \leq L(x,y,w) \\ & \leq L(x,y,v) (\text{since we chose } v \text{ to maximize } L(x,y,w)) \\ & = MTC(x,y,v) \end{aligned}$$

Hence, v maximizes $MTC(x,y,w)$ over $w \in [q,t]$, so by Lemma 6.1 v is the bottleneck of $[q,t]$. Thus, to compute $FindBot([s,t])$ we just compare v and $FindBot([s,q])$. Note that this is not technically a pure binary search, since we do not just recurse to the half-interval $[s,q]$. Rather, we recurse to a half-interval and a single vertex.

Say that $L(x,y,v) \leq R(x,y,v)$. Then, by lemma 6.2, $MTC(x,y,v) = L(x,y,v)$. But we chose v to maximize $L(x,y,v)$, so v must be the hardest vertex to avoid on all of $[q,t]$. More formally, note that for any $w \in [q,t]$ we have $MTC(x,y,w) \leq L(x,y,w) \leq L(x,y,v) = MTC(x,y,v)$. Hence, v is the bottleneck of $[q,t]$, so to compute $FindBot([s,t])$ we just compare v and $FindBot([s,q])$. Note that this is not technically a pure binary search, since we do not just recurse to the half-interval $[s,q]$. Rather, we recurse to a half-interval and a single vertex.

Alternatively, say that $L(x,y,v) > R(x,y,v)$. Then, letting π be the shortest *centered* path avoiding v, we know that π goes through c_y but not c_x (lemma 6.2). But this means that π avoids v and c_x, so it avoids all of $[c_x,v]$, so it certainly avoids $[s,q]$ (figure 4). In particular, v must be at least as hard to avoid as any vertex in $[s,q]$, so it is a better candidate for the bottleneck than any vertex in $[s,q]$. More formally, if $w \in [s,q]$ then π is a path through c_y that avoids w, so $MTC(x,y,w) \leq w(\pi) = R(x,y,v) = MTC(x,y,v)$. Thus, the bottleneck cannot be in $[s,q]$, so $FindBot([s,t]) = FindBot([q,t])$.

In either case, it takes constant time recurse to an interval of half the size (and possibly check another single vertex). Thus, it takes $O(\log(n))$ time to find the bottleneck vertex for $CI[x,y,i]$, which leads to an overall running time of $\widetilde{O}(n^2)$ for finding all the bottleneck vertices.

Figure 5: Pseudo Code for the algorithm FindBot in section 6

```
Input: An interval $I = [s,t] \subseteq CI[x,y,i]$
Output: A vertex $w = \text{argmax}_{v \in I} MTC(x,y,v)$
0. IF $|[s,t]| \leq 2$
       Return $\text{argmax}_{w \in [s,t]}(MTC(x,y,w))$
1. $q \leftarrow \lfloor (s+t)/2 \rfloor$
2. $v \leftarrow v_L(x,y,[q,t])$
3. IF $L(x,y,v) \leq R(x,y,v)$
       w ← FindBot([s,q])
       to-output ← $\text{argmax}_{v,w}\{MTC(x,y,v), MTC(x,y,w)\}$
4. IF $L(x,y,v) > R(x,y,v)$
       to-output ← FindBot([q,t])
5. Return to-output
```

7. CONCLUSION

We have presented a deterministic distance sensitivity oracle with O(1) query time, $\widetilde{O}(\mathrm{n}^2)$ space requirement, and $\widetilde{O}(\mathrm{mn})$ construction time. We cannot really hope to improve upon the static version, but can we make the oracle dynamic: if we delete a single vertex, can we do better than constructing another oracle from scratch? Also, can we efficiently handle more than one vertex failure at a time? Finally, can we achieve better results by settling for approximate shortest paths?

8. REFERENCES

[1] M.O. Ball, B.L. Golden, and R.V. Vohra. Finding the most vital arcs in a network. *Operations Research Letters*, 8:73Ű76, 1989.

[2] A. Bar-Noy, S. Khuller, and B. Schieber. *The complexity of finding most vital arcs and nodes.* Technical Report No CS-TR-3539, Institute for Advanced Studies, University of Maryland, College Park, MD, 1995.

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

[4] A. Bernstein and D. Karger. Improved distance sensitivity oracles via random sampling. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '08), San Francisco, California*, pages 34-43, 2008.

[5] R.A. Chowdhury and V. Ramach. Improved distance oracles for avoiding link-failure. In *Proc. of the 13th International Symposium on Algorithms and Computation (ISAAC '02), Vancouver, Canada*, LNCS 2518, pages 523-534, 2002.

[6] C. Demetrescu, M. Thorup, R.A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a node or link failure. *SIAM Journal of computing*, 37(5):1299-1318, 2008.

[7] E.Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269-271, 1959.

[8] Y. Emek, D. Peleg, and L. Roditty. A near-linear time algorithm for computing replacement paths in planar directed graphs. In *Proc. of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '08), San Francisco, California*, pages 428-435, 2008.

[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.

[11] J. Hershberger, S. Suri, and A. Bhosle. On the difficulty of some shortest path problems. In *Proc. of the 20th International Symposium of Theoretical Aspects of Computer Science (STACSŠ03), Berlin, Germany*, pages 343Ű354, 2003.

[12] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. *Proceedings of the 40th IEEE Annual Symposium on Foundations of Computer Science (FOCS '99), New York, NY*, pages 81-89, 1999.

[13] L. Roditty and U. Zwick. Replacement Paths and k Simple Shortest Paths in Unweighted Directed Graphs. In *Proc. of the 32nd International Colloquium on Automata, Languages and Programming, Lisboa, Portugal*, pages 249-260, 2005.