Oracles for Distances Avoiding a Failed Node or Link *

Camil Demetrescu † Mikkel Thorup ‡ R. A. Chowdhury §
Vijaya Ramachandran ¶

Abstract

We consider the problem of preprocessing an edge-weighted directed graph G to answer queries that ask for the length of a shortest path from any given vertex x to another given vertex y avoiding an arbitrary vertex or edge. As a natural application, this problem models routing in networks subject to node or link failures. We describe a deterministic oracle with constant query time for this problem that uses $O(n^2 \log n)$ space, where n is the number of vertices in G. We also show that if we are willing to use $\Theta(n^{2.5})$ space, we can reduce the preprocessing time by a factor of $\sqrt{n}$ while maintaining constant query time. Our algorithms can find the shortest path avoiding a failed node or link in time proportional to the length of the path.

AMS Subject Classifications. [05C85, 68W01] Graph algorithms; [68P05] Data structures; [05C38] Shortest paths; [90B18] Network failures.

---

*Part of the results presented in this article for link failures appeared in the Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02) and in the Proceedings of the 13th International Symposium on Algorithms and Computation (ISAAC'02).

†Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", via Salaria 113, 00198 Rome, Italy. Email: demetres@dis.uniroma1.it. URL: http://www.dis.uniroma1.it/~demetres. FAX: +39-06-85300849. Author supported in part by the Italian Ministry of University and Scientific Research (Project "ALGO-NEXT: Algorithms for Internet and the Web").

‡AT&T Labs-Research, 180 Park Avenue, Florham Park, NJ 07932. Email: mthorup@research.att.com. URL: http://www.research.att.com/info/mthorup.

§The University of Texas at Austin, Department of Computer Sciences, Austin, TX 78712. Email: shaikat@cs.utexas.edu. URL: http://www.cs.utexas.edu/~shaikat. Supported in part by an MCD Graduate Fellowship and NSF Grant CCR-9988160.

¶The University of Texas at Austin, Department of Computer Sciences, Austin, TX 78712. Email: vlr@cs.utexas.edu. URL: http://www.cs.utexas.edu/~vlr. Supported in part by Texas Advanced Research Program Grant 003658-0029-1999 and NSF Grant CCR-9988160.

1 Introduction

In the *distance sensitivity* problem, we wish to construct a data structure (which we call *distance sensitivity oracle*) for an edge-weighted directed graph G that supports any sequence of the following queries:

`v-dist`(x, y, v): returns the distance from vertex x to vertex y in $G - \{ v \}$, i.e., the length (sum of edge weights) of the shortest possible path from x to y in G avoiding vertex v, if one exists, and $+\infty$ otherwise.

`e-dist`(x, y, u, v): returns the distance from vertex x to vertex y in $G - \{ (u, v) \}$, i.e., the length (sum of edge weights) of the shortest possible path from x to y in G avoiding edge (u, v), if one exists, and $+\infty$ otherwise.

We also consider the corresponding path queries, which we denote by `v-path`(x, y, v) and `e-path`(x, y, u, v), respectively. In this article, we denote by n the number of vertices and by m the number of edges in G. We also assume that edge weights are non-negative. In our bounds, space is measured as number of memory words, where each word can hold the label of a vertex or edge or the weight of an edge or path.

1.1 Motivation

Our motivating scenario is a network where node/link failures happen quite rarely. As soon as a node or link failure has been noticed, we want to be able to answer distance queries and provide directions for shortest paths in the network without the failed node or link. On the other hand, we assume that we have plenty of time to compute a new data structure in the background. We model a network as a weighted directed graph where vertices correspond to network nodes, and edges correspond to network links. In this scenario, `v-dist`(x, y, v) yields the distance from node x to node y in the network avoiding failed node v, and `e-dist`(x, y, u, v) yields the distance from node x to node y in the network avoiding failed link (u, v).

We note that the ability to deal with node/link failures enables us to deal with some other related aspects of the network. For example, by dealing with a link failure (u, v), we actually deal with arbitrary changes to its weight. More precisely, we can simply compute the distance from vertex x to vertex y in the graph where (u, v) has its weight changed to w as $\min\{\texttt{e-dist}(x, y, u, v), d_{xu} + w + d_{vy}\}$, where d_{xu} is the distance from vertex x to vertex u and d_{vy} is the distance from vertex v to vertex y in the graph. Here, a weight change could model that traffic is moving slower/faster along a certain link. An interesting application of dealing with single weight changes is a local search like the one in [9]. There, one wants to consider a neighborhood of a given weight setting, where each neighbor is obtained by changing a single weight.

Another motivation for solving the distance sensitivity problem arises from recent interest in Vickrey pricing of networks [11, 19]. We describe this application in more detail in Section 1.4.

1.2 Related work

A variant of the distance sensitivity problem, related to reachability in directed acyclic graphs with respect to edge failures, was first introduced by King and Sagert in [14], where they consider the problem of supporting *sensitivity queries* of the kind: "Is there a path from vertex x to vertex y that does not contain edge (u, v)?" Here, we are concerned with general weighted digraphs and distance queries instead of reachability queries, and consider both vertex and edge failures.

This problem is similar to the *replacement paths* problem ([11], also see the erratum for this paper, and [12, 18, 21]) which, given a pair of vertices x and y in G, computes the set of shortest paths from x to y avoiding each of the vertices (or edges) on π_{xy} one at a time, where π_{xy} is the original shortest path from x to y in G. A method for solving this problem on undirected graphs in $O(m + n \log n)$ preprocessing time and $O(n)$ space is given for the vertex removal case in [18] and for the edge removal case in [11]. However, in this paper we are interested in finding replacement paths for all possible vertex pairs and for the more general case of a directed graph.

Of a similar flavor is the *most vital node* (or *arc*) problem [1, 2, 4, 18], which is the problem of identifying the vertex (or edge) on a given shortest path, whose removal results in the longest replacement path.

The most natural approach to the distance sensitivity problem would be to use one of the recent *dynamic* all pairs shortest paths (APSP) algorithms [5, 6, 13, 15], and delete the failed vertex or edge. The best bounds ([6, 24]) take $\widetilde{O}(n^2)$ amortized time for real weighted directed graphs, but then queries avoiding the failed vertex or edge are answered in constant time. However, our goal here is to answer a query as quickly as possible after a vertex or edge failure, and then it may be faster to compute the answer from scratch at each query using an $O(n \log n + m)$ single-source shortest paths (SSSP) algorithm [10].

Another extreme solution would be to construct a table that for each vertex pair (x, y) and each vertex/edge stores the distance from x to y avoiding that vertex/edge. For vertex failures, such a table of size $O(n^3)$ is trivially computed by n APSP computations. For real-weighted sparse graphs, this takes $O(mn^2 + n^3 \log \log n)$ time using a recent $O(mn + n^2 \log \log n)$ time APSP algorithm [20], and for dense graphs, it takes $O(n^4 \sqrt{\log \log n} / \log n)$ time using a very recent $O(n^3 \sqrt{\log \log n} / \log n)$ time APSP algorithm [25] (see also [22]). However, for edge failures the size of the table is $\Theta(mn^2)$ and requires m APSP computations. Space can be reduced (at least for the edge failure case) by working from one source x at a time, constructing a shortest paths tree $T(x)$. This tree changes only if we remove any of the $O(n)$ vertices or edges in it. Hence, it is only for these vertices and edges that we need to record new distances from x to all other vertices $y \in V$. As a result, the total space requirement is $O(n^3)$ for both vertex and edge deletion. The query time is still $O(1)$. As elaborated in [23], it is also easy to get a running time of $O(mn^2 + n^3 \log \log n)$. However, we consider the cubic space prohibitive for most practical applications.

1.3 Our results

The main result of this article is a deterministic oracle with fast query time for both vertex and edge failures that uses nearly the same space as that required for storing the distance matrix of the input graph. More precisely, we construct an oracle that uses $O(n^2 \log n)$ space and answers distance queries subject to a vertex or edge failure in $O(1)$ worst-case time. This result is quite surprising, since the space bound is significantly smaller than $\Theta(n^3)$ and yet our scheme answers queries in $O(1)$ time. We also present an $\Omega(m)$ space lower bound for the single-source version of the problem. Since m can be as high as $\Omega(n^2)$, our oracle is thus almost space-optimal.

The construction time for our oracle is $O(mn^2 + n^3 \log n)$ in the worst case. However, if one is willing to settle for $\Theta(n^{2.5})$ space, we can improve the preprocessing time to $O(mn^{1.5} + n^{2.5} \log n)$, and the query time remains constant. In Table 1 we place our bounds in perspective by comparing them to the bounds for related problems obtainable with previous algorithms: in the context of most vital node detection and Vickrey pricing, we are extrapolating the performance of algorithms designed for a single source-destination pair to the all-pairs case.

To achieve our bounds, we construct data structures where we store information about all pairs shortest paths excluding only vertices or edges with specific properties, rather than excluding all possible vertices or edges. We also store information about shortest paths where we exclude vertices on entire subpaths, rather than single vertices. We remark that our algorithms are very simple, and thus amenable to efficient implementations.

Part of the results presented in this paper for link failures were presented in two conference papers: one by the first two authors [7] which presented the two algorithms, and the other by the last two authors [3]. The paper [3] improves some of the results in [7], but it also has an extra claim on construction time which is incorrect. Additionally, this paper presents results obtained by the last two authors on extending the oracles in [7, 3] to efficiently handle node failures in addition to link failures. In the process, the presentation of the results has been changed from that in the conference papers: we first present algorithms to handle node failures directly and then extend these algorithms to deal with link failures. This paper also presents a lower bound result obtained by the first two authors on the space requirement for the single source version of the distance oracle problem.

1.4 Vickrey pricing in networks

The Vickrey mechanism is a generalization of the *sealed bid second price* auction, in which the highest bidder wins the auction but pays a price equal to the second highest bid. This auction protocol motivates a rational bidder to bid truthfully [17]. In a distributed network in which multiple rational self-interested agents own different parts of the network, Vickrey mechanism is often the best way to determine the utility of various network elements. In order to elicit truthful responses from the agents, each agent is compensated in proportion to the marginal utility he/she brings to the network. Willing manipulations by participating agents is eliminated by making an agent's payment depend only on the declarations of

References	Context	Graph Type	Construction Time	Space	Query Time
Nardelli et al. [18]	Most Vital Node Detection [`v-dist`(x, y, v)]	Undirected	$O(mn^2 + n^3 \log n)$	$O(n^3)$	$O(1)$
Hershberger & Suri [11]	Vickrey Pricing in Networks [`e-dist`(x, y, u, v)]	Undirected	$O(mn^2 + n^3 \log n)$	$O(n^3)$	$O(1)$
Our Contribution	Vertex/Edge Failure [`v-dist`(x, y, v)] [`e-dist`(x, y, u, v)]	Directed (or Undirected)	Method 1		
			$O(mn^2 + n^3 \log n)$	$O(n^2 \log n)$	$O(1)$
			Method 2		
			$O(mn^{1.5} + n^{2.5} \log n)$	$O(n^{2.5})$	$O(1)$

Table 1: Known results and our contribution.

other agents.

Consider a scenario in which we need to find shortest paths in a network G, where links are owned by self-interested agents. Agents are assumed to bid on each link individually. Nisan and Ronen [19] formulated the following expression as the payment $p^e(x, y)$ to be made to the owner of a link e for a given vertex pair (x, y):

$$p^e(x,y) = \begin{cases} d_{xy}(G|_{w_e=\infty}) - d_{xy}(G|_{w_e=0}) & if\ e \in \pi_{xy} \\ 0 & otherwise \end{cases}$$

where π_{xy} is a shortest path from vertex x to vertex y in G, and $d_{xy}(G|_{w_e=k})$ is the distance from vertex x to vertex y in G where the weight of edge e is set to k. For any $e \in \pi_{xy}$, the term $d_{xy}(G|_{w_e=0})$ can be simply computed as $d_{xy}(G|_{w_e=0}) = d_{xy} - w_e$, where d_{xy} is the distance from x to y in G. However, computing $d_{xy}(G|_{w_e=\infty})$ naïvely for all $e \in \pi_{xy}$ requires running an $O(m + n \log n)$ time shortest paths algorithm [8, 10] on $G - \{e\}$ for each $e \in \pi_{xy}$. This can be as high as $O(mn + n^2 \log n)$ in the worst case. This problem was studied in [11], but no improvement to this trivial bound is known.

The distance sensitivity problem we study in this article is a generalization of the above problem to the situation where one is potentially interested in finding all Vickrey payments for all vertex pairs (instead of a single pair). In this case our first algorithm can carry out the entire computation using significantly less space than that used by the naïve algorithm. On the other hand, our second algorithm reduces both time and space requirements of the computation. The complexities of all three algorithms are compared in Table 2. Observe, however, that for very sparse graphs the running time of the naïve algorithm can be improved to $O(mn(m + n \log \log n))$ [20].

Algorithm	Time	Space	Query time
Naïve	$O(n^2(m + n\log n))$	$O(n^3)$	$O(1)$
Our 1st Result	$O(n^2(m + n\log n))$	$O(n^2\log n)$	$O(1)$
Our 2nd Result	$O(n^{1.5}(m + n\log n))$	$O(n^{2.5})$	$O(1)$

Table 2: Complexity of computing Vickrey payments for all vertex pairs.

1.5 Organization of the article

The remainder of this article is organized as follows. In Section 2 we introduce the notation used in the article and we discuss some simple properties that will be useful in the description of our results. In particular, we define the notion of "path cover", showing how to use information about shortest paths that avoid all vertices on certain paths in the graph to determine a shortest path that avoids a single vertex. In Section 3 we show how to efficiently compute shortest paths from any given vertex to all other vertices in a directed graph G with non-negative real-valued edge weights where we avoid all vertices on certain types of paths. These tools are used in Section 4 to devise an oracle for the distance sensitivity problem that answers `v-dist` queries in constant worst-case time using nearly the same space required for storing a single distance matrix. This oracle can be constructed in $O(mn^2 + n^3 \log n)$ worst-case time. In Section 5 we show that, if one is willing to settle for more space, we can reduce the preprocessing time to $O(mn^{1.5} + n^{2.5}\log n)$. This second oracle uses $O(n^{2.5})$ space while still answering `v-dist` queries in constant worst-case time. Section 6 shows how to extend the oracles designed for vertex failures to deal also with edge failures and Section 7 addresses the problem of supporting path queries `v-path` and `e-path`. A space lower bound for the single-source version of the distance sensitivity problem is discussed in Section 8. Finally, Section 9 provides some concluding remarks.

2 Preliminaries

Let $G = (V, E, w)$ be a directed graph with vertex set V, edge set E, and edge weight function w. Throughout the article, we assume that, for each pair of vertices x and y such that y is reachable from x, there is a unique shortest path from x to y. This is without loss of generality, since ties can be broken arbitrarily (see, e.g., [6]).

Notation	Meaning
G	edge-weighted directed graph $G = (V, E, w)$
w_{xy}	weight of edge (x, y) in G
p_{xy}	path $\langle x, v_1, v_2, \cdots, v_{k-1}, y\rangle$ from vertex x to vertex y in G
$p_{xy} \cdot p_{yz}$	concatenation of path p_{xy} with path p_{yz}
$w(p_{xy})$	length of path p_{xy} (sum of weights of edges in p_{xy})
π_{xy} or $[x, y]$	shortest path from vertex x to vertex y in G (we assume it is unique)
$]x, y[$	$\pi_{xy} - \{ x, y \}$
$[x, y[$	$\pi_{xy} - \{ y \}$
$]x, y]$	$\pi_{xy} - \{ x \}$
h_{xy}	number of edges in π_{xy}
d_{xy}	distance from vertex x to vertex y in G $\quad (d_{xy} = w(\pi_{xy}))$
$T(x)$	shortest path tree rooted at x in G (x is at level 0 in $T(x)$)
$\widehat{G}$	reversed graph $(V, \widehat{E}, \widehat{w})$ s.t. $\widehat{E} = \{(x, y) : (y, x) \in E\}$ and $\widehat{w}_{xy} = w_{yx}$
$\widehat{T}(x)$, $\widehat{\pi}_{xy}$	$T(x)$ and π_{xy} in $\widehat{G}$ instead of G
$B_T(i, j)$	set of all paths in tree T connecting vertices at level i to vertices at level j
$a < b$ in p_{xy}	vertex a precedes vertex b in p_{xy}
π^{uv}_{xy}	shortest path from x to y in $G - [u, v]$
$d_{xy}(l_1, l_2, r_1, r_2)$	$\min_{\substack{a \in [l_1, l_2[\\ b \in]r_1, r_2]}} \{ d_{xa} + w(\pi^{l_2 r_1}_{ab}) + d_{by} \}$ (see Definition 2.1)

Table 3: Notation used in the article.

2.1 Notation

In this article, a path p_{xy} is a sequence of vertices of the form $p_{xy} = \langle v_0, v_1, v_2, \cdots, v_{k-1}, v_k\rangle$ such that $v_0 = x$, $v_k = y$, and $(v_i, v_{i+1}) \in E$, for every i, $0 \leq i < k$. Thus, $G - p_{xy}$ is the subgraph of G induced by the vertex set $V - \{x, v_1, \cdots, v_{k-1}, y\}$. We let $p_{xy} \cdot p_{yz}$ denote the concatenation of path p_{xy} with path p_{yz}. We denote by w_{xy} the weight of edge (x, y) in G, and we indicate with $w(p_{xy})$ the length of p_{xy}, i.e., the sum of weights of edges in p_{xy}. We also denote by $T(x)$ the single-source shortest path tree of G with source x and we denote by π_{xy} the (unique) shortest path from vertex x to vertex y in G, if any. Using a geometrical analogy, we sometimes look at shortest paths as "segments", using notation $[x, y]$ instead of π_{xy}. Similarly, we sometimes use $]x, y[$ to denote $\pi_{xy} - \{ x, y \}$, $[x, y[$ to denote $\pi_{xy} - \{ y \}$, and $]x, y]$ to denote $\pi_{xy} - \{ x \}$. We indicate with d_{xy} the length of π_{xy} (distance from x to y in G), and with h_{xy} the number of edges in π_{xy} (number of hops). If y is not reachable from x, we assume $d_{xy} = h_{xy} = +\infty$. By $\widehat{G}$ we denote the directed graph obtained from G by reversing the orientation of edges. Thus, $\widehat{\pi}$ and $\widehat{T}$ denote π and T in $\widehat{G}$. If T is a rooted tree, we let $B_T(i, j)$ denote the set of paths in T that connect vertices at level i with vertices at level $j > i$ in the tree, assuming that the root of T has level 0. Notice that, if $\pi_{cd} \in B_{T(x)}(i, j)$ then $h_{cd} = j - i$. Let a and b be vertices on p_{xy}; we say that $a < b$ if a appears before b on p_{xy}, and $a \leq b$ if $a < b$ or $a = b$. Finally, by π^{uv}_{xy} we denote a shortest path from vertex x to vertex y in $G - [u, v]$. The path π^{uv}_{xy} could be thought of as an optimal "replacement path" from x to y to be used in case all vertices in $[u, v]$ fail. Observe that, for each internal vertex v in π_{xy}, $\texttt{v-dist}(x, y, v) = w(\pi^{vv}_{xy})$. Indeed, the goal of this article is to provide methods for answering queries about π^{vv}_{xy}. The notation used in this article is summarized in Table 3.

2.2 Structural properties

By our assumption of uniqueness of shortest paths, for any pair of vertices $a, b \in \pi_{xy}$ such that $a \leq b$, π_{ab} is a subpath of π_{xy}. Moreover, if $\pi_{xy} = \{v_0, v_1, \ldots, v_{k-1}, v_k\}$ then $\hat{\pi}_{yx} = \{v_k, v_{k-1}, \ldots, v_1, v_0\}$, i.e., one is the reversal of the other.

We now discuss a simple structural property of π^{uv}_{xy}. In particular, the following claim shows that, if y is reachable from x in $G - [u, v]$, then π^{uv}_{xy} and π_{xy} share a common prefix and a common suffix, and are vertex-disjoint elsewhere (see Figure 1). Intuitively, the internal subpath of π^{uv}_{xy} that is vertex-disjoint from π_{xy} can be thought of as an "optimal detour" that avoids vertices in $[u, v]$:

Claim 1 *Let $G = (V, E, w)$ be an edge-weighted directed graph. For any $x, y \in V$ and for any $u, v \in \pi_{xy}$ with $x < u \leq v < y$, if $\pi^{uv}_{xy} \neq \emptyset$, then there exist two vertices $a, b \in \pi_{xy}$ such that $\pi^{uv}_{xy} = \pi_{xa} \cdot p_{ab} \cdot \pi_{by}$, where $p_{ab} \cap \pi_{ab} = \{a, b\}$.*

Proof. We first notice that π_{xy} and π^{uv}_{xy} are both paths in G, since every path in $G - [u, v]$ is also a path in G. Moreover, $\pi_{xy} \neq \pi^{uv}_{xy}$, since $u \in \pi_{xy}$, but $u \notin \pi^{uv}_{xy}$.

Now, let p_{xa} be the longest common prefix of π^{uv}_{xy} and π_{xy}, and let p_{by} be their longest common suffix. These subpaths are never empty, since π^{uv}_{xy} and π_{xy} share at least their end-points. This proves the existence of vertices a and b in our claim.

Since every subpath of π_{xy} is a shortest path (by the optimal substructure property of shortest paths) and shortest paths are unique in G, then $p_{xa} = \pi_{xa}$ and $p_{by} = \pi_{by}$. Furthermore, since $\pi_{xy} \neq \pi^{uv}_{xy}$, then $a \neq b$ and $p_{xa} \cap p_{by} = \emptyset$. Thus, we can write π^{uv}_{xy} as $\pi_{xa} \cdot p_{ab} \cdot \pi_{by}$ for some p_{ab}.

It remains to prove that $p_{ab} \cap \pi_{ab} = \{a, b\}$. Suppose by contradiction that there is a vertex $c \in p_{ab} \cap \pi_{ab}$ such that $a < c < b$. Now, observe that $c \notin [u, v]$, since $p_{ab} \cap [u, v] = \emptyset$, and that $[u, v]$ is a subpath of π_{ab}, since $\pi_{xa} \cap [u, v] = \emptyset$ and $\pi_{by} \cap [u, v] = \emptyset$. Assume without loss of generality that $c < u$ in π_{xy} (the case $v < c$ is completely analogous). Consider the subpath π_{ac} of π_{ab}, and notice that it is a shortest path in both G and $G - [u, v]$. Now, p_{ab} is a shortest path in $G - [u, v]$ and thus, by the optimal-substructure property, its subpath p_{ac} has to be shortest as well in $G - [u, v]$. Since shortest paths are unique in $G - [u, v]$, then $p_{ac} = \pi_{ac}$, and thus π_{xy} and π^{uv}_{xy} share the same subpath from x to $c > a$. This means that a cannot be the last vertex of the longest common prefix of π^{uv}_{xy} and π_{xy}, clearly a contradiction. $\square$

2.3 Path covering

We now discuss the notion of "path covering" that will be crucial to proving the correctness of our query algorithms.

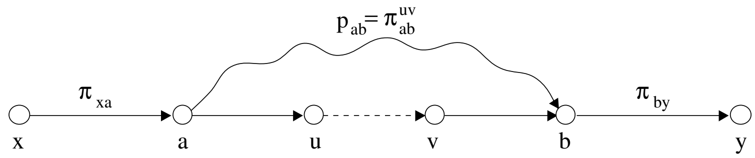


Figure 1: Structure of a replacement path $\pi_{xy}^{uv} = \pi_{xa} \cdot p_{ab} \cdot \pi_{by}$.

Definition 2.1 *Let $x \leq l_1 < l_2 \leq r_1 < r_2 \leq y$ be vertices on π_{xy}. We define $d_{xy}(l_1, l_2, r_1, r_2)$ as follows:*

$$d_{xy}(l_1, l_2, r_1, r_2) = \min_{\substack{a \in \ [l_1, l_2[\\ b \in \ ]r_1, r_2]}} \{ d_{xa} + w(\pi_{ab}^{l_2 r_1}) + d_{by} \}$$

and we say that a value d covers *$[l_1, l_2[\ \times \ ]r_1, r_2]$ w.r.t. x, y if $d \leq d_{xy}(l_1, l_2, r_1, r_2)$. In this case, we also say that d* covers *all paths of the form $\pi_{xa} \cdot \pi_{ab}^{l_2 r_1} \cdot \pi_{by}$ for each $a \in \ [l_1, l_2[$ and for each $b \in \ ]r_1, r_2]$.*

Observe that $d_{xy}(l_1, l_2, r_1, r_2)$ is the distance from vertex x to vertex y in G using paths that first follow a shortest path from x to some vertex a in $[l_1, l_2[$, then take an optimal detour that avoids all vertices in $[l_2, r_1]$, and finally go through a shortest path from some vertex b in $]r_1, r_2]$ to y. The following claim, which easily follows from Definition 2.1 and from Claim 1, states that, if $l_1 = x$ and $r_2 = y$, then $d_{xy}(l_1, l_2, r_1, r_2)$ is equal to the length of the shortest path from x to y avoiding $[u, v]$:

Claim 2 *Let $x < u \leq v < y$ be vertices on π_{xy}. Then $w(\pi_{xy}^{uv}) = d_{xy}(x, u, v, y)$.*

We now show how information about the distance from x to y avoiding $[u, v]$ with detours having constrained positions of their end-points can be used to compute $w(\pi_{xy}^{uv})$. The following claim will be useful to prove the correctness of our query algorithms.

Claim 3 *For any $x < \widehat{u} \leq \widetilde{u} < u \leq v < \widetilde{v} \leq \widehat{v} < y$ on π_{xy},*

$$d_{xy}(x, u, v, y) \ = \ \min\{d_{xy}(x, \widetilde{u}, \widetilde{v}, y), \ d_{xy}(\widehat{u}, u, v, y), \ d_{xy}(x, u, v, \widehat{v})\}$$

Proof. Let the end-points of the optimal detour in π_{xy}^{uv} be a and b, $a < b$, and consider all possible positions of a in $[x, u[$:

- a in $[x, \widehat{u}[$: $d_1 = d_{xy}(x, u, v, \widehat{v})$ handles the cases when b lies in $]v, \widehat{v}]$, i.e., d_1 covers $[x, \widehat{u}[\ \times \ ]v, \widehat{v}]$ w.r.t. x, y. Moreover, $d_2 = d_{xy}(x, \widetilde{u}, \widetilde{v}, y)$ handles the cases when b lies in $]\widehat{v}, y]$, i.e., d_2 covers $[x, \widehat{u}[\ \times \ ]\widehat{v}, y]$ w.r.t. x, y. Thus, $d_3 = \min\{d_{xy}(x, u, v, \widehat{v}), \ d_{xy}(x, \widetilde{u}, \widetilde{v}, y)\}$ handles all possible positions of b in $]v, y]$, i.e., d_3 covers $[x, \widehat{u}[\times]v, y]$ w.r.t. x, y.

- a in $[\widehat{u}, u[$: $d_4 = d_{xy}(\widehat{u}, u, v, y)$ handles the cases where b lies in $]v, y]$, i.e., d_4 covers $[\widehat{u}, u[\ \times \ ]v, y]$ w.r.t. x, y.

Thus, for each possible position of a in $[x, u[$, $d_5 = \min(d_{xy}(x, \tilde{u}, \tilde{v}, y),\ d_{xy}(\hat{u}, u, v, y),\ d_{xy}(x, u, v, \hat{v}))$ handles all possible positions of b in $]v, y]$, i.e., d_5 covers $[x, u[\ \times\ ]v, y]$ w.r.t. x, y. Since d_5 equals the length of some path from x to y in $G - [u, v]$, we can then conclude that $d_5 = d_{xy}(x, u, v, y)$. □

3 Distances under deletion of paths

In this section we provide simple algorithms for computing distances in a directed graph G with non-negative real-valued edge weights where we avoid all vertices on certain paths. These algorithms will be useful in Section 4 and Section 5 for constructing distance sensitivity oracles.

Let x be a vertex and let P be a set of shortest paths in G. We consider the problem of designing a procedure $\texttt{exclude}(G, x, P)$ that computes for each path $\pi \in P$ the distances from vertex x to all other vertices in $G - \pi$. Throughout this article we assume that deletion of the vertices on a given path π results in the deletion of all its vertices including its end-points.

We can compute $\texttt{exclude}(G, x, P)$ with a straight-forward algorithm that runs in $O(|P|(m + n \log n))$ worst-case time using a Dijkstra computation [10] on the graph G with all vertices in π deleted, for each $\pi \in P$. In the remainder of this section we show that this computation can be made considerably more efficient if we restrict our attention to $P \subseteq T(x)$ (i.e., every path in P is also a path in $T(x)$), and if we assume that paths in P are "independent", a notion we define in Definition 3.1.

3.1 Algorithm `fast-exclude`

In this section we devise a variant of $\texttt{exclude}(G, x, P)$, which we call $\texttt{fast-exclude}(G, x, P)$, for the case when $P \subseteq T(x)$. As above, our goal is to compute for each path $\pi \in P$ the distances from vertex x to all other vertices in $G - \pi$.

Let $P \subseteq T(x)$, and for any path $\pi \in P$, denote by $T_x(\pi)$ the subtree of $T(x)$ rooted at the first vertex of π, and let W_π be the set of all vertices in $T_x(\pi)$ except the vertices on π. We observe that only vertices in W_π may have their distances from x increased if we remove from G the vertices on π. Now, consider the following directed graph $G_\pi = (V_\pi, E_\pi, w_\pi)$, where:

- $V_\pi = W_\pi \cup \{x\}$.
- E_π contains an edge from x to each vertex in W_π and the edges in G induced by vertices in W_π.
- $w_{\pi,ab}$ is the weight of edge (a, b) in G_π defined as:

$$w_{\pi,ab} = \begin{cases} \min_{c \notin T_x(\pi)}\{d_{xc} + w_{cb}\} & \textit{if } a = x \\ w_{ab} & \textit{otherwise} \end{cases}$$

where we assume that $d_{xc} = +\infty$ if c is not reachable from x in G and $w_{cb} = +\infty$ if (c, b) is not an edge of G.

It is not difficult to see (see proof of Claim 4) that the shortest path from vertex x to a vertex v in W_π has the same length in $G - \pi$ as the shortest path from x to v in G_π. Hence, distances in $G - \pi$ from x to all vertices in W_π can be computed by a Dijkstra computation on G_π with source x. The algorithm `fast-exclude`(G, x, P) works in the same way as `exclude`(G, x, P), but it uses G_π instead of G for each π in P.

Since the graph G_π is typically smaller than G, `fast-exclude` can be expected to have better performance than `exclude` for any collection of paths $P \subseteq T(x)$. We now define the notion of *independent shortest paths*, and we show that `fast-exclude` performs significantly better than `exclude` when $P \subseteq T(x)$ is a collection of independent shortest paths.

Definition 3.1 *Let T be a rooted tree and let π_{uv} and $\pi_{u'v'}$ be two paths in T. We say that π_{uv} and $\pi_{u'v'}$ are* independent *in T if the subtree of T rooted at u and the subtree of T rooted at u' are disjoint.*

Claim 4 *If $P \subseteq T(x)$ is a set of pairwise independent shortest paths in $T(x)$, then algorithm* `fast-exclude`(G, x, P) *requires $O(m + n \log n)$ time in the worst case and computes the same output as* `exclude`$(G,\ x, P)$.

Proof. Using the notation given above, for a given $\pi \in P$, let π'_{xy} be the shortest path from x to any $y \in W_\pi$ avoiding the vertices on π. Let b be the first vertex on π'_{xy} such that $b \in W_\pi$ and let c be the vertex preceding b. Then let us write $\pi'_{xy} = p_{xc} \cdot \langle c, b\rangle \cdot \pi'_{by}$ where $\langle c, b\rangle$ is the path from c to b formed by the single edge (c, b). Since for any $z \notin T_x(\pi)$, the path π_{xz} is composed entirely of the vertices in $T(x) - T_x(\pi)$, it follows that $p_{xc} = \pi_{xc}$. Also note that π'_{by} cannot contain any vertex $z \notin T_x(\pi)$, since if it contains such a vertex z then π_{xz} will be a shorter path to z contradicting our choice of b. These two observations justify the use of G_π instead of $G - \pi$ in order to compute the shortest path tree rooted at x avoiding the vertices on π.

For each $\pi \in P$, $|V_\pi|$ is never greater than the number of vertices in $T_x(\pi)$. Since the paths in P are pairwise independent, any two such subtrees are disjoint for distinct paths in P. Since each E_π contains edges in G induced by vertices in W_π and edges from x to only the vertices in W_π, it follows that the sum of the cardinalities of all V_π and E_π are linear in n and m, respectively. Hence, `fast-exclude`(G, x, P) runs in $O(m + n \log n)$ time when $P \subseteq T(x)$ is a set of independent shortest paths. $\square$

4 Oracle with $O(1)$ query time and $O(n^2 \log n)$ space

In this section we describe a deterministic oracle for single vertex failure with constant query time that uses nearly the same space as that required for storing a single distance matrix. In particular, we show how to preprocess a graph with non-negative real-valued edge weights in $O(mn^2 + n^3 \log n)$ worst-case time, producing a compact oracle that uses $O(n^2 \log n)$ space and answers `v-dist` queries in $O(1)$ worst-case time.

4.1 Data structure

Using $O(n^2 \log n)$ space, we maintain each d_{xy} and h_{xy}, and we maintain six matrices dl, dr, sl, sr, vl, and vr of size $n \times n \times \lfloor \log_2 n \rfloor$ defined for every pair of distinct vertices x and y as follows:

- $dl[x, y, i]$ = distance from vertex x to vertex y in $G - \pi$, where π is the sub-path of π_{xy} starting at level 2^{i-1} and ending at level $2^i - 1$ in $T(x)$, and $1 \leq i \leq \log_2 h_{xy}$;
- $dr[x, y, i]$ = distance from vertex y to vertex x in $\widehat{G} - \pi$, where π is the sub-path of $\widehat{\pi}_{yx}$ starting at level 2^{i-1} and ending at level $2^i - 1$ in $\widehat{T}(y)$, and $1 \leq i \leq \log_2 h_{xy}$;
- $sl[x, y, i]$ = distance from vertex x to vertex y in $G - \{v\}$, where v is the vertex of π_{xy} at level 2^{i-1} in $T(x)$, and $1 \leq i < 1 + \log_2 h_{xy}$;
- $sr[x, y, i]$ = distance from vertex y to vertex x in $\widehat{G} - \{v\}$, where v is the vertex of $\widehat{\pi}_{yx}$ at level 2^{i-1} in $\widehat{T}(y)$, and $1 \leq i < 1 + \log_2 h_{xy}$;
- $vl[x, y, i]$ = vertex of π_{xy} at level 2^{i-1} in $T(x)$, where $1 \leq i \leq 1 + \log_2 h_{xy}$;
- $vr[x, y, i]$ = vertex of $\widehat{\pi}_{yx}$ at level 2^{i-1} in $\widehat{T}(y)$, where $1 \leq i \leq 1 + \log_2 h_{xy}$.

4.2 Preprocessing

The above quantities are computed in the preprocessing phase as follows:

- Distances d_{xy} and h_{xy} and matrices vl and vr are easily initialized from shortest path trees of G.
- To compute $dl[x, y, i]$, we call procedure $\texttt{exclude}(G, x, B_{T(x)}(2^{i-1}, 2^i - 1))$, discussed in Section 3, for each x and for each i, $1 \leq i < \log_2 n$.
- To compute $dr[x, y, i]$, we call procedure $\texttt{exclude}(\widehat{G}, y, B_{\widehat{T}(y)}(2^{i-1}, 2^i - 1))$, for each y and for each i, $1 \leq i < \log_2 n$.
- For each x and for each i, $1 \leq i < 1 + \log_2 (n - 1)$, we compute $sl[x, y, i]$ by calling procedure $\texttt{fast-exclude}(G, x, B_{T(x)}(2^{i-1}, 2^{i-1}))$.
- We compute $sr[x, y, i]$ by calling procedure $\texttt{fast-exclude}(\widehat{G}, y, B_{\widehat{T}(y)}(2^{i-1}, 2^{i-1}))$, for each y and for each i, $1 \leq i < 1 + \log_2 (n - 1)$.

function `v-dist`$(x, y, v) : \mathcal{R}$

1. **if** $d_{xv} + d_{vy} > d_{xy}$ **then return** d_{xy}
2. $l \leftarrow \lceil \log_2 h_{xv} \rceil$
3. **if** $l = \log_2 h_{xv}$ **then return** $sl[x, y, l+1]$
4. $r \leftarrow \lceil \log_2 h_{vy} \rceil$
5. **if** $r = \log_2 h_{vy}$ **then return** $sr[x, y, r+1]$
6. $\hat{u} \leftarrow vr[x, v, l], \hat{v} \leftarrow vl[v, y, r]$
7. $d \leftarrow \min\{d_{x\hat{u}} + sl[\hat{u}, y, l], sr[x, \hat{v}, r] + d_{\hat{v}y}\}$
8. **if** $h_{xv} \leq h_{vy}$ **then** $d \leftarrow \min\{d, dl[x, y, l]\}$ {v in left half}
9. **else** $d \leftarrow \min(d, dr[x, y, r])$ {v in right half}
10. **return** d

Figure 2: Query algorithm for the first oracle.

4.3 Query

The query algorithm is shown in Figure 2. We only address the general interesting case where $v \neq x$ and $v \neq y$, otherwise the answer is clearly $+\infty$. In line 1 of the algorithm, we get rid of the case where $v \notin \pi_{xy}$ and return d_{xy} as the answer. Lines 2 and 3 take care of the case where v is 2^l edges away from vertex x on π_{xy} for some non-negative integer l, $0 \leq l < \log_2 h_{xy}$. Lines 4 and 5 handle the case where v is 2^r edges away from vertex y on $\widehat{\pi}_{yx}$ for some non-negative integer r, $0 \leq r < \log_2 h_{xy}$. Lines 6 to 9 take care of the remaining cases.

4.4 Analysis

We first discuss the correctness of our query procedure. Using the matrices sl and sr, lines 2 to 5 of the query algorithm answer the following two types of trivial queries: (1) $h_{xv} = 2^l$ for some non-negative integer l, $0 \leq l < \log_2 h_{xy}$, and (2) $h_{vy} = 2^r$ for some non-negative integer r, $0 \leq r < \log_2 h_{xy}$. So in order to prove the correctness of `v-dist`, we need only to prove the correctness of the code segment of lines 6 to 9 that handles the nontrivial case when neither of the above two conditions hold.

In line 7, we assign d to the minimum of $d_{x\hat{u}} + sl[\widehat{u}, y, l]$ and $sr[x, \widehat{v}, r] + d_{\hat{v}y}$, where $d_{x\hat{u}} + sl[\widehat{u}, y, l] = d_{xy}(\widehat{u}, v, v, y)$ and $sr[x, \widehat{v}, r] + d_{\hat{v}y} = d_{xy}(x, v, v, \widehat{v})$. In line 8, we consider the case where $h_{xv} \leq h_{vy}$, i.e., v is in the first half of π_{xy} (see Figure 3). In this case, $0 < h_{x\hat{u}} < h_{\hat{u}v} = 2^{l-1} \leq h_{v\hat{v}}$. The value $dl[x, y, l]$ is the distance from x to y avoiding a sub-path having 2^{l-1} vertices and starting at a vertex that is 2^{l-1} edges away from x on π_{xy}. Let the end-points of that sub-path be $\widetilde{u}$ and $\widetilde{v}$, and $h_{x\tilde{u}} < h_{x\tilde{v}}$. So, we have, $x < \widehat{u} \leq \widetilde{u} < v < \widetilde{v} \leq \widehat{v} < y$. Thus, by Claim 3, d covers $[x, v[\times]v, y]$ following the assignment in line 8. By construction of matrices dl, sr, and sr, d is always equal to the weight of some path from x to y that does not use vertex v. Thus, we can conclude that d is the desired answer to `v-dist`(x, y, v). A similar argument holds for the case where $h_{xv} > h_{vy}$ (line 9).

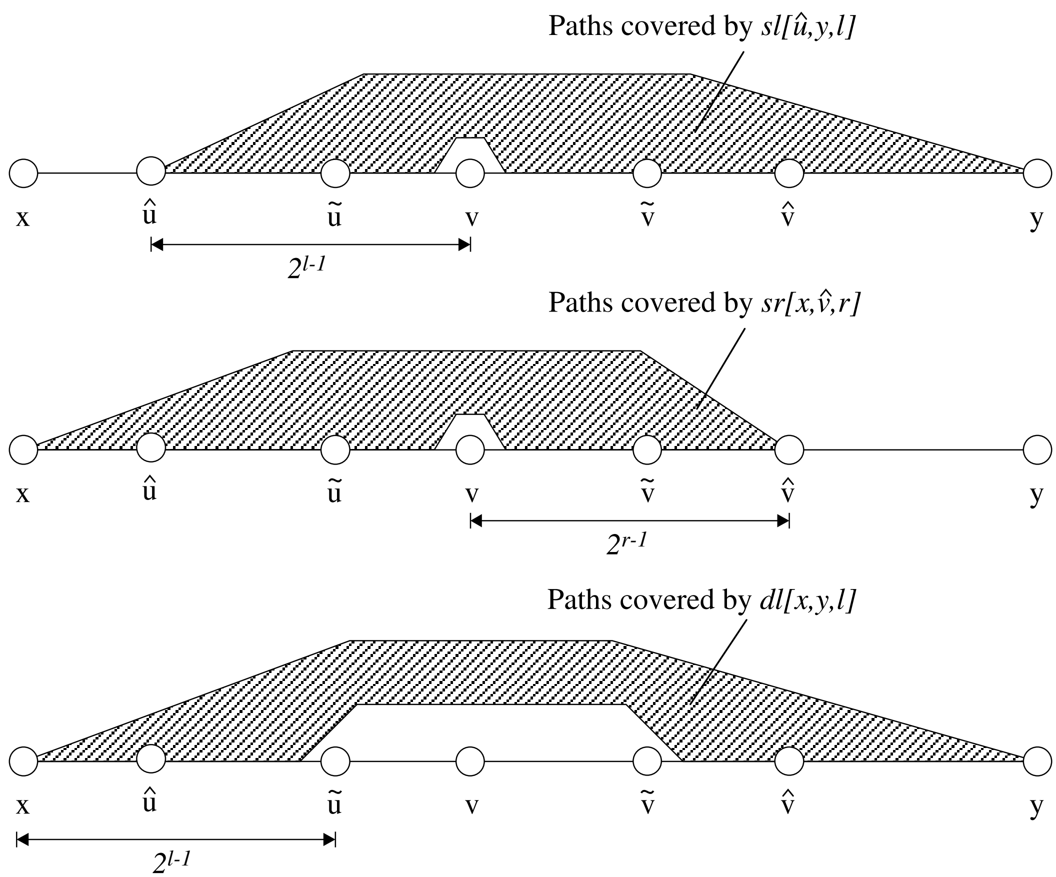


Figure 3: Query example with $h_{xv} \leq h_{vy}$: The distance from vertex x to vertex y in $G-\{v\}$ can be obtained by taking the minimum of $d_{x\hat{u}} + sl[\hat{u}, y, l]$, $sr[x, \hat{v}, r] + d_{\hat{v}y}$, and $dl[x, y, l]$. Notice that the union of the paths in the grey areas in the figure is the set of all possible detours avoiding vertex v.

We now address the running times of preprocessing and query procedures.

Claim 5 *Preprocessing requires $O(mn^2 + n^3 \log n)$ worst-case time and any* `v-dist` *operation requires $O(1)$ worst-case time.*

Proof. We observe that the number of distinct paths in $B_{T(x)}(2^{i-1}, 2^i - 1)$ is exactly the same as the number of vertices on level $2^i - 1$ in $T(x)$. Hence, the total number of distinct paths in all $B_{T(x)}(2^{i-1}, 2^i - 1)$, for $1 \leq i < \log_2 n$, is bounded from above by the number of vertices in $T(x)$. Since `exclude`(G, x, P) runs in $O(|P|(m + n \log n))$ time and $\sum_{1 \leq i < \log_2 n} |B_{T(x)}(2^{i-1}, 2^i - 1)| = O(n)$, the matrices dl and dr can be calculated in $O(mn^2 + n^3 \log n)$ worst-case time. On the other hand, for each x and for each i, $1 \leq i < 1 + \log_2 (n-1)$, we can compute $sl[x, y, i]$ by calling procedure `fast-exclude`$(G, x, B_{T(x)}(2^{i-1}, 2^{i-1}))$ since the single-vertex paths in $B_{T(x)}(2^{i-1}, 2^{i-1})$ are trivially pairwise independent in $T(x)$. Since `fast-exclude`$(G, x, B_{T(x)}(2^{i-1}, 2^{i-1}))$ runs in $O(m + n \log n)$, the total time required to compute the matrix sl is $O(mn \log n + n^2 \log^2 n)$. Similarly the matrix sr can be computed in $O(mn \log n + n^2 \log^2 n)$ time. Hence the preprocessing time is dominated by the time to compute the dl and dr matrices and requires $O(mn^2 + n^3 \log n)$ worst-case time.

Since the query algorithm executes a constant number of steps, it runs in $O(1)$ worst-case time. □

5 Improving the preprocessing time

In this section we show that, if one is willing to settle for more space, we can design a distance sensitivity oracle where we reduce the preprocessing time to $O(mn^{1.5} + n^{2.5} \log n)$. This second oracle uses $O(n^{2.5})$ space and answers distance queries in $O(1)$ worst-case time.

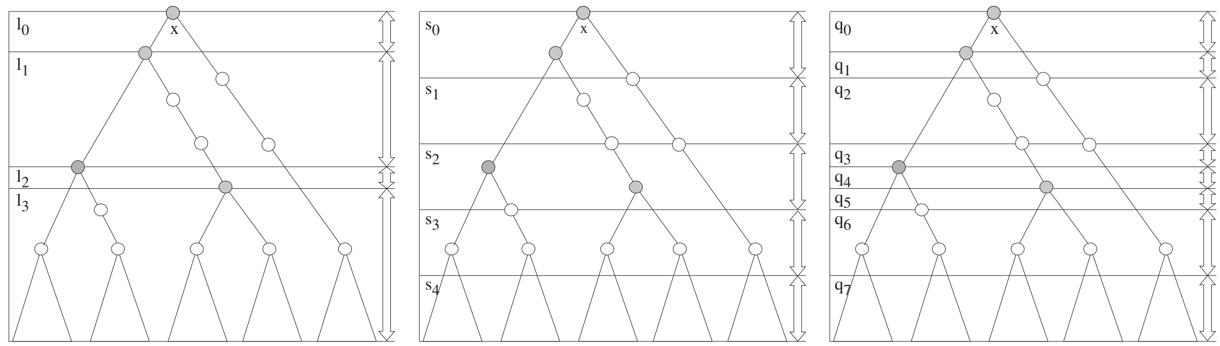
5.1 Data structure

We maintain each d_{xy} and h_{xy}, and we maintain five matrices dh, dt, vc, dc, and bc using $O(n^{2.5})$ space. Matrix dh has size $n \times n \times \lfloor \sqrt{n} \rfloor$, matrices dt, vc, and dc have size $n \times n \times \lfloor 2\sqrt{n} \rfloor$, and matrix bc has size $n \times n$. They are defined as follows:

- $dh[x, y, i]$ = distance from vertex x to vertex y in $G - \{v\}$, where v is the vertex of π_{xy} at level i in $T(x)$ and $0 < i \leq \sqrt{n}$;
- $dt[x, y, i]$ = distance from vertex x to vertex y in $G - \{v\}$, where v is the vertex of π_{xy} at level i in $\widehat{T}(y)$ and $0 < i \leq 2\sqrt{n}$;
- $vc[x, y, i]$ = vertex of π_{xy} at level q_i in $T(x)$, where $q_0 = 0 < q_1 < \cdots < q_k < n$ is any increasing sequence of $k+1 \leq 2\sqrt{n}$ positive numbers depending on x, and $q_i - q_{i-1} \leq \sqrt{n}$ for any i, $1 \leq i \leq k$ (a method for obtaining the sequence $\{q_i : 0 \leq i \leq k\}$ is described in Section 5.2.1);
- $dc[x, y, i]$ = distance from vertex x to vertex y in $G - [\hat{u}, \hat{v}]$, where $\hat{u}$ is the successor of $vc[x, y, i]$ in π_{xy} and $\hat{v} = vc[x, y, i+1]$, if $h_{\hat{v}y} > \sqrt{n}$, and undefined otherwise;
- $bc[x, y]$ = index i such that $q_i + 1 \leq h_{xy} \leq q_{i+1}$.

5.2 Preprocessing

As distances d_{xy} and h_{xy} are easily initialized from shortest path trees of G, we focus on constructing matrices dh, dt, dc, vc, and bc. Since bands $B_{T(x)}(i, i)$ contain paths formed by single vertices, which are trivially pairwise independent in $T(x)$, constructing matrix dh can be done by performing calls to algorithm `fast-exclude`(G, x, $B_{T(x)}(i,\ i)$), presented in Section 3, for each vertex x and for each i such that $0 < i \leq \sqrt{n}$. Similarly, matrix dt can be initialized via calls to algorithm `fast-exclude`($\widehat{G}$, y, $B_{\widehat{T}(y)}(i,\ i)$) for each vertex y and for each i such that $0 < i \leq 2\sqrt{n}$.



(a) Sequence $\{l_i\}$ obtained by cutting T'(x) at vertices of degree >1

(b) Sequence $\{s_i\}$ obtained by cutting at regular intervals of height $\sqrt{n}$

(c) Sequence $\{q_i\}$ obtained by merging sequences $\{l_i\}$ and $\{s_i\}$

Figure 4: Constructing matrix dc: Cutting tree $T'(x)$ to form bands of pairwise independent shortest paths in it.

To compute dc, we consider the problem of cutting each shortest path tree $T(x)$ into at most $2\sqrt{n}$ bands of height $\le \sqrt{n}$, finding a suitable subset of each band containing pairwise independent shortest paths in $T(x)$, and calling `fast-exclude` to compute distances without those paths. To do so, we need to compute for each vertex x a suitable sequence $\{q_i : 0 \le i \le k\}$.

5.2.1 Computing the q_i's

For each vertex x we wish to find a sequence $q_0 = 0 < q_1 < \cdots < q_k < n$ of length $k+1 \le 2\sqrt{n}$ such that $q_i - q_{i-1} \le \sqrt{n}$ for any i, $1 \le i \le k$, and compute a subset of paths in $B_{T(x)}(q_i+1, q_{i+1})$ that are pairwise independent in $T(x)$. The following claim provides a nice combinatorial property on trees that helps us solve the problem.

Let T be a rooted directed tree with n vertices and let $size(v)$ be the number of vertices in the subtree of T rooted at v. Let T' be obtained from T by deleting any vertex u such that $size(u) \le \sqrt{n}$ in T.

Claim 6 *For any directed tree T with n vertices, at most $\sqrt{n}$ vertices in T' have out-degree > 1.*

Proof. Since T' contains only vertices that have size greater than $\sqrt{n}$ in T, T' has at most $\sqrt{n}$ leaves. This implies that at most $\sqrt{n}$ vertices of T' have out-degree > 1. □

Let $l_0 < l_1 < \cdots < l_k$ be the sequence of levels in T' such that at each level l_i there is a vertex with out-degree > 1 in T' (Figure 4a). By Claim 6, $k \le \sqrt{n}$. We now observe that cutting T' at each l_i yields bands of vertex-disjoint paths:

Claim 7 *For any i, $1 \le i < k$, $B_{T'}(l_i+1, l_{i+1})$ is a band of vertex-disjoint paths.*

Proof. $B_{T'}(l_i + 1, l_{i+1})$ contains all paths that connect vertices at level $l_i + 1$ with vertices at level l_{i+1} in T'. The proof easily follows by observing that, by the definition of sequence $\{l_i\}$, all vertices in T' at levels $l_i + 1$ to $l_{i+1} - 1$ have out-degree ≤ 1. $\square$

Notice that, since by Claim 7 $B_{T'}(l_i + 1, l_{i+1})$ is a band of vertex-disjoint paths and all of them start at the same level $l_i + 1$ in T', then they are clearly pairwise independent in T'. As T' is obtained by pruning T, $B_{T'}(l_i + 1, l_{i+1}) \subseteq B_T(l_i + 1, l_{i+1})$ and paths in $B_{T'}(l_i + 1, l_{i+1})$ are also pairwise independent in T. Unfortunately, however, we are not guaranteed that $l_i - l_{i-1} \leq \sqrt{n}$, as we need for constructing dc. However, we note that splitting a band of vertex-disjoint paths yields again bands of vertex-disjoint paths. This leads to the following claim:

Claim 8 *If $B_T(i + 1, j)$ is a band of vertex-disjoint paths, then for any $i < h < j$, both $B_T(i + 1, h)$ and $B_T(h + 1, j)$ are bands of vertex-disjoint paths.*

Let $s_0 < s_1 < \cdots < s_{\lfloor\sqrt{n}\rfloor}$ be a sequence such that $s_i = i \cdot \lfloor\sqrt{n}\rfloor$ (Figure 4b). By Claim 8, if we merge sequences $\{l_i\}$ and $\{s_i\}$ and get rid of duplicates, we obtain an ordered sequence $\{q_i\}$ of length at most $2\sqrt{n}$ with the desired properties (Figure 4c).

5.2.2 Computing vc, dc, and bc

We remark that $\{q_i\}$ induces at most $2\sqrt{n}$ bands of vertex-disjoint paths in $T'(x)$ with height $\leq \sqrt{n}$. Clearly, these paths are pairwise independent in $T'(x)$. To initialize dc, we can thus perform calls to `fast-exclude`$(G, x, B_{T'(x)}(q_i + 1, q_{i+1}))$ for each vertex x and for each $0 < i \leq 2\sqrt{n}$. Again, we can use `fast-exclude` instead of `exclude`.

At this point, one may argue that excluding only independent paths in $T'(x)$, which is obtained by pruning $T(x)$, might not give the correct result for some $dc[x, y, i]$ if $y \notin T'(x)$. However, $dc[x, y, i]$ is defined only when $h_{\hat{v}y} > \sqrt{n}$, where $\hat{v} = vc[x, y, i + 1]$, and $\hat{v} \in T'(x)$ in this case. Thus $dc[x, y, i]$ is correctly computed by calling `fast-exclude`$(G, x, B_{T'(x)}(q_i + 1, q_{i+1}))$.

Finally, we observe that once sequences $\{q_i\}$ have been computed for each $T(x)$, matrices vc and bc can be easily initialized.

5.3 Query

The query algorithm is shown in Figure 5. We first get rid of the cases where $v \notin \pi_{xy}$ and $h_{vy} \leq 2\sqrt{n}$, for which the answers are stored explicitly in d_{xy} and $dt[x, y, h_{vy}]$, respectively (lines 1–2). In line 3 we retrieve the unique index i such that v falls in $B_{T(x)}(q_i + 1, q_{i+1})$, and then in lines 4–5 we identify the vertices $\widehat{u}$ and $\widehat{v}$ on the path π_{xy} in $T(x)$ that are at levels $q_i + 1$ and q_{i+1}, respectively. The correct answer is given in line 6 by accessing matrices dc, dh, and dt.

function `v-dist`$(x, y, v) : \mathcal{R}$

1. **if** $d_{xv} + d_{vy} > d_{xy}$ **then return** d_{xy}
2. **if** $h_{vy} \leq 2\sqrt{n}$ **then return** $dt[x, y, h_{vy}]$
3. $i \leftarrow bc[x, v]$
4. $\hat{u} \leftarrow$ successor of $vc[x, y, i]$ in π_{xy}
5. $\hat{v} \leftarrow vc[x, y, i+1]$
6. $d \leftarrow \min\{\, dc[x, y, i],$ $d_{x\hat{u}} + dh[\hat{u}, y, h_{\hat{u}v}],$ $dt[x, \hat{v}, h_{v\hat{v}}] + d_{\hat{v}y} \,\}$
7. **return** d

Figure 5: Query algorithm for the second oracle.

5.4 Analysis

To prove the correctness of `v-dist` in the case that lines 3–7 are executed, we first note that $h_{vy} > 2\sqrt{n}$ implies $h_{\hat{v}y} > \sqrt{n}$, since by construction $q_i - q_{i-1} \leq \sqrt{n}$, and thus $dc[x, y, i]$ is well-defined. We now prove that the answer takes into account all possible configurations of the end-points of detours π_{ab}^{uv} (see Figure 1). It is easy to see that $x < \widehat{u} < v < \widehat{v} < y$ and:

- $dc[x, y, i] = d_{xy}(x, \widehat{u}, \widehat{v}, y)$
- $d_{x\hat{u}} + dh[\hat{u}, y, h_{\hat{u}v}] = d_{xy}(\widehat{u}, v, v, y)$
- $dt[x, \hat{v}, h_{v\hat{v}}] + d_{\hat{v}y} = d_{xy}(x, v, v, \widehat{v})$

Thus, by Claim 3, $d = d_{xy}(x, v, v, y)$.

Claim 9 *Preprocessing requires $O(mn^{1.5} + n^{2.5} \log n)$ worst-case time and any* `v-dist` *operation requires $O(1)$ worst-case time.*

Proof. Growing shortest path trees $T(x)$ for all vertices x requires $O(mn + n^2 \log n)$ time in the worst-case [10]. The proof for the preprocessing follows from Claim 4 by observing that initializing dc, dh, and dt is carried out via $O(\sqrt{n})$ calls to `fast-exclude` for each vertex x. The bound for queries is straightforward. □

6 Handling edge failures

The oracles in the previous two sections can be easily extended to handle edge failures by maintaining one additional matrix de of size $n \times n$ for any x and y:

- $de[x, y]$ = distance from vertex x to vertex y in G without the first edge of π_{xy}.

function `e-dist`$(x, y, u, v) : \mathcal{R}$

1. **if** $d_{xu} + w_{u,v} + d_{vy} > d_{xy}$ **then return** d_{xy}
2. $d_1 \leftarrow$ `v-dist`(x, y, u)
3. $d_2 \leftarrow d_{xu} + de[u, y]$
4. $d \leftarrow \min\{d_1, d_2\}$
5. **return** d

Figure 6: Query algorithm for link failure.

Claim 10 *The matrix de can be initialized in $O(mn + n^2 \log n)$ time.*

Proof. The proof directly follows from the properties of an earlier version of the algorithm `fast-exclude` in [7] based on the notion of *edge-independent paths*. However, since in this article we use the notion of *vertex-independent paths* instead, we present a proof of the claim based on it.

Consider a given $T(x)$ and let $v_1, v_2, \cdots, v_k$ be the children of x in $T(x)$. We extend $T(x)$ to $T'(x)$ and thus G to G' by introducing k new vertices $u_1, u_2, \cdots u_k$ and for $1 \le i \le k$, replacing each edge (x, v_i) in $T(x)$ by two consecutive edges (x, u_i) and (u_i, v_i). Clearly removing any vertex u_i from $T'(x)$ has the same effect as removing the corresponding edge (x, v_i) from $T(x)$. Therefore, since the single vertex paths in $B_{T'(x)}(1, 1)$ are trivially independent, we can compute $de[x, y]$ for all $y \in V - \{x\}$ in time $O(m + n \log n)$ by calling `fast-exclude`$(G', x, B_{T'(x)}(1, 1))$. (Note that we can perform the same computation in the same time bound without constructing G' explicitly, but instead extending `fast-exclude` to handle this special case.) Since we need to call `fast-exclude` once for each $x \in V$, the total initialization time for de is $O(mn + n^2 \log n)$. $\square$

6.1 Query

The query algorithm is shown in Figure 6. In line 1 of the algorithm, we get rid of the case where the failed edge is not on π_{xy}. In line 2, d_1 is assigned the distance from x to y avoiding vertex u and thus edge (u, v). In line 3, d_2 is assigned the distance from x to y that avoids the edge (u, v) but passes through the vertex u. In line 4, d is assigned the minimum of d_1 and d_2 which is then returned in line 5 as the shortest x to y distance avoiding (u, v).

6.2 Analysis

We observe that the paths in G from x to y that avoid (u, v) can be divided into two groups: (1) paths that avoid u and (2) paths that pass through u. Line 2 of the algorithm finds the length of the shortest path in group (1) and line 3 does the same for group (2). Thus the minimum of the two distances obtained in lines 2 and 3 gives the required distance. Note that since `v-dist` runs in constant worst-case time, so does `e-dist`.

function `v-path`$(x, y, v) : E$

if $d_{xv} + d_{vy} > d_{xy}$ **then return** first edge of π_{xy}
$l \leftarrow \lceil \log_2 h_{xv} \rceil$
if $l = \log_2 h_{xv}$ **then return** $sle[x, y, l+1]$
$r \leftarrow \lceil \log_2 h_{vy} \rceil$
if $r = \log_2 h_{vy}$ **then return** $sre[x, y, r+1]$
$\hat{u} \leftarrow vr[x, v, l], \hat{v} \leftarrow vl[v, y, r]$
$d \leftarrow \min\{d_{x\hat{u}} + sl[\hat{u}, y, l], sr[x, \hat{v}, r] + d_{\hat{v}y}\}$
if $d = d_{x\hat{u}} + sl[\hat{u}, y, l]$ **then** $e \leftarrow$ first edge of $\pi_{x\hat{u}}$
else $e \leftarrow sre[x, \hat{v}, r]$
if $h_{xv} \leq h_{vy}$ **and** $dl[x, y, l] < d$ **then** $e \leftarrow dle[x, y, l]$ {v in left half}
if $h_{xv} > h_{vy}$ **and** $dr[x, y, r] < d$ **then** $e \leftarrow dre[x, y, r]$ {v in right half}
return e

Figure 7: Path version of the query algorithm for our first oracle.

6.3 Avoiding two consecutive edges

In this section we observe that the distance from any vertex x to another vertex y avoiding two *consecutive* failed edges on π_{xy}, can be computed in constant time by maintaining another $n \times n$ matrix $\widehat{de}$, which is the dual of de in $\widehat{G}$.

- $\widehat{de}[x, y]$ = distance from vertex y to vertex x in $\widehat{G}$ without the first edge of $\hat{\pi}_{yx}$.

Assuming that the two consecutive failed links are (u, v) and (v, w), all paths in G from x to y avoiding those two edges can be divided into two groups: (1) paths that avoid v: the length of the shortest such path can be found by calling `v-dist`(x, y, v), and (2) paths that pass through v: the length of the shortest such path is given by $\widehat{de}[x, v] + de[v, y]$. Thus the smaller of these distances is the required distance.

7 Supporting path queries

The oracles presented in this paper can be easily extended to support path queries of the form `v-path`(x, y, v) and `e-path`(x, y, u, v), which return the first edge on the shortest path from vertex x to vertex y in $G - \{v\}$ and in $G - \{(u, v)\}$, respectively.

In this section, we show how to extend the oracle given in Section 4 to support `v-path`(x, y, v) queries. Extending the other oracles and supporting `e-path`(x, y, u, v) queries can be easily done in a similar way. We add to the data structure of Section 4.1 the following additional matrices:

- $dle[x, y, i] = (x, x')$, where (x, x') is the first edge on the shortest path from vertex x to vertex y ($\neq x$) in $G - \pi$, and π is the sub-path of π_{xy} starting at level 2^{i-1} and ending at level $2^i - 1$ ($1 \leq i \leq \log_2 h_{xy}$) in $T(x)$;

- $dre[x, y, i] = (x, x')$, where (x', x) is the last edge on the shortest path from vertex y to vertex x ($\neq y$) in $\widehat{G} - \pi$, and π is the sub-path of $\widehat{\pi}_{yx}$ starting at level 2^{i-1} and ending at level $2^i - 1$ ($1 \leq i \leq \log_2 h_{xy}$) in $\widehat{T}(y)$;

- $sle[x, y, i] = (x, x')$, where (x, x') is the first edge on the shortest path from vertex x to vertex y ($\neq x$) in $G - \{v\}$, and v is the vertex of π_{xy} at level 2^{i-1} ($1 \leq i < 1 + \log_2 h_{xy}$) in $T(x)$;

- $sre[x, y, i] = (x, x')$, where (x', x) is the last edge on the shortest path from vertex y to vertex x ($\neq y$) in $\widehat{G} - \{v\}$, and v is the vertex of $\widehat{\pi}_{yx}$ at level 2^{i-1} ($1 \leq i < 1 + \log_2 h_{xy}$) in $\widehat{T}(y)$;

Matrices *dle*, *dre*, *sle*, and *sre* can be easily initialized in the preprocessing phase within the same time bounds by a simple extension of procedures `exclude` and `fast-exclude` described in Section 3. Figure 7 shows an implementation of operation `v-path` obtained as a modification of the query procedure `v-dist` given Figure 2. The analysis is straightforward and is left to the reader.

8 A space lower bound

In this section, we briefly discuss a space lower bound for the single-source version of the distance sensitivity problem. This version is relevant, e.g., in routing in networks, where routers typically only need to know which outgoing link to choose to get on a shortest path to a packet's destination [16]. Without edge failures, this can be represented in $O(n)$ space with a single-source shortest path tree. In Sections 4 and 6, we have seen that one can represent all pairs shortest paths with link failures in $O(n^2 \log n)$ space. A natural question is whether there is something to do for the single-source case. If d is the maximal number of hops to the destination, in [23] it is shown that one can store distances avoiding a failed link in $O(d \cdot n)$ space.

We now discuss the negative result that for any number m of edges in a graph G, we may need $\Omega(m)$ space to represent distances with edge failures. The graph is as follows. Consider a chain over the vertices $v_0, \ldots, v_{n-1}$, where v_0 is the source and n is the number of vertices of G. Each edge (v_i, v_{i+1}) has weight $2 \cdot n$. If an edge (v_i, v_j), $j > i + 1$, is in G, it gets weight $(4 \cdot n - i - j) \cdot n + id(i, j)$, where $id(i, j) < n$. Hence, if link (v_i, v_{i+1}) fails and we want to go to v_j from v_0, the unique optimal replacement link to use is (v_i, v_j), if it is in G. This holds even in the undirected case. The graph has up to n replacement edges, and for each of them, there is at least a link failure and a destination that will need it. Hence, a link failure oracle will have to know the edges of G, and will need to know all the $id(i, j)$ values. This yields $\Omega(m)$ space, which might be as high as $\Omega(n^2)$ for dense graphs.

9 Conclusions

We have presented compact data structures for maintaining information about shortest paths in a weighted directed graph in case of both vertex failures and edge failures. We have shown that, surprisingly, such a data structure can be stored using nearly the same space required to store a single distance matrix, while still supporting queries in constant time. Our oracle can be easily constructed in $O(mn^2 + n^3 \log n)$ time, matching the preprocessing time of all pairs variants of similar problems such as most vital node detection [18] and Vickrey pricing [11] in networks; while these algorithms require $\Theta(n^3)$ space, our oracle requires only $O(n^2 \log n)$ space. Furthermore, we have shown that using $O(n^{2.5})$ space we can improve construction time to $O(mn^{1.5} + n^{2.5} \log n)$.

Our oracles are different from the case of dynamic algorithms, where distances have to be updated after each vertex or edge failure. Instead, our oracles are already prepared to answer distance queries following the failure of any single vertex or edge, and so the delay time in answering a query is minimized. If failures in a network happen quite rarely, when a node or link goes down we have time to construct a new oracle in the background to cope with a possible additional failure. It would be interesting to explore whether compact oracles with fast query time that are able to deal with more than one failure at a time can be constructed. Finally, can we further improve construction time?

References

[1] M.O. Ball, B.L. Golden, and R.V. Vohra. Finding the most vital arcs in a network. *Operations Research Letters*, 8:73–76, 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] R.A. Chowdhury and V. Ramachandran. Improved distance oracles for avoiding a link-failure. In *Proc. of the 13th International Symposium on Algorithms and Computation (ISAAC'02), Vancouver, Canada, LNCS 2518*, pages 523–534, 2002.

[4] H.W. Corley and D.Y. Sha. Most vital links and nodes in weighted networks. *Operations Research Letters*, 8:157–160, 1982.

[5] C. Demetrescu and G.F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada*, 2001.

[6] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the Association for Computing Machinery* (JACM), 51(6), pp. 968-992, November 2004.

[7] C. Demetrescu and M. Thorup. Oracles for distances avoiding a link-failure. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02), San Francisco, California*, pages 838–843, 2002.

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

[9] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proceedings of the 19th IEEE INFOCOM - The Conference on Computer Communications, Tel-Aviv, Israel*, pages 519–528, 2000.

[10] 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.

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

[12] 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 (STACS'03), Berlin, Germany*, pages 343–354, 2003.

[13] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99), New York City, New York*, pages 81–99, 1999.

[14] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences* (JCSS), 65(1), pages 150-167, 2002.

[15] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Computing and Combinatorics Conference (COCOON'01), Guilin, China, LNCS 2108*, pages 268–277, 2001.

[16] J. Moy. OSPF: Anatomy of an internet routing protocol. Addison-Wesley, 1999.

[17] A. Mas-Collel, W. Whinston, and J. Green. Microeconomic theory. *Oxford University Press*, 1999.

[18] E. Nardelli, G. Proietti, and P. Widmayer. Finding the most vital node of a shortest path. In *Proceedings of the 7th Annual International Computing and Combinatorics Conference (COCOON'01), Guilin, China, LNCS 2108*, pages 278–287, 2001.

[19] N. Nisan and A. Ronen. Algorithmic mechanism design. In *Proceedings of the 31st Annual ACM Symposium in Theory of Computation (STOC'99)*, pages 129–140, 1999.

[20] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1), pages 47–74, 2003.

[21] L. Roditty and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. To appear in the *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05), Lisboa, Portugal*, July 2005.

[22] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 30:309–318, 1998.

[23] M. Thorup. Fortifying OSPF/IS-IS against link-failure, 2001. Manuscript.

[24] M. Thorup. Fully-Dynamic all-pairs shortest paths: faster and allowing negative cycles In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, pp. 384–396, 2004.

[25] U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. In *Proc. of the 15th International Symposium on Algorithms and Computation (ISAAC'04), HongKong, China, LNCS 3341*, pages 921–932, 2004.