Dual-Failure Distance and Connectivity Oracles*

Ran Duan
University of Michigan

Seth Pettie
University of Michigan

Abstract

Spontaneous failure is an unavoidable aspect of all networks, particularly those with a physical basis such as communications networks or road networks. Whether due to malicious coordinated attacks or other causes, failures temporarily change the topology of the network and, as a consequence, its connectivity and distance metric. In this paper we look at the problem of efficiently answering connectivity, distance, and shortest route queries in the presence of two node or link failures. Our data structure uses $\tilde{O}(n^2)$ space and answers queries in $\tilde{O}(1)$ time, which is within a polylogarithmic factor of optimal and nearly matches the single-failure distance oracles of Demestrescu *et al.* It may yet be possible to find distance/connectivity oracles capable of handling any fixed number of failures. However, the sheer complexity of our algorithm suggests that moving beyond dual-failures will require a fundamentally different approach to the problem.

1 Introduction

We consider the problem of answering distance queries in a weighted directed graph $G = (V, E, \ell)$, where one or more nodes or edges are unavailable due to failure or other causes. Specifically, given source and target vertices x, y and a set $F \subset V$, the problem is to report $\delta_{G-F}(x, y)$, where $\delta_{G'}$ is the distance function w.r.t. the subgraph G'.[1] In the absence of failure, the best oracle for answering distance queries in $O(1)$ time is a trivial $n \times n$ lookup table, where $n = |V|$ is the number of vertices. Thus, a distance oracle that is sensitive to node failures should be considered (nearly) optimal if it occupies (nearly) quadratic space and answers queries in (nearly) constant time. Demetrescu et al. [6] showed that single-failure distance queries can be answered in constant time by an oracle occupying $O(n^2 \log n)$ space. Very recently Bernstein and Karger [3] improved the construction time of [6] from $\tilde{O}(mn^2)$ to $\tilde{O}(n^2\sqrt{m})$. Both [6] and [3] highlighted the problem of finding distance oracles capable of dealing with more than one failure.

In this paper we show that dual-failure distance queries can be answered in $O(\log n)$ time using $O(n^2 \log^3 n)$ space. Our data structure and query algorithm are considerably more complex than those of [6, 3], an unfortunate situation which is partially explained by a sharp qualitative difference between shortest paths avoiding 1 and 2 failures. If p is a shortest path from x to y and u a failed vertex, the shortest path avoiding u consists of a prefix of p followed by a "detour" avoiding p (and u), followed by a suffix of p. In the presence of 2 or more failures it is no longer possible to create such a clean decomposition. The shortest path avoiding two failures on p may depart from and return to p an unbounded number of times and furthermore, each such segment may not even be a shortest path. With 3 (or more) failures the potential complexity of the optimal detours becomes even more unwieldy. Our main result is as follows:

Theorem 1.1. *Given a weighted directed graph $G = (V, E, \ell)$, where $\ell : E \to \mathbb{R}$ assigns arbitrary real lengths, a data structure with size $O(n^2 \log^3 n)$ can be constructed in polynomial time such that given vertices $x, y \in V$ and two failed vertices or edges $u, v \in V \cup E$, $\delta_{G-\{u,v\}}(x, y)$ can be reported in $O(\log n)$ time. Furthermore, a path with this length can be returned in $O(\log n)$ time per edge.*

As a special case, Theorem 1.1 allows one to answer connectivity queries in $O(\log n)$ time. We only prove Theorem 1.1 for two vertex failures. There is a simple reduction from an f-edge failure distance query to $O(1)$ f-vertex failure queries, for any fixed f. We can use Theorem 1.1 to answer distance queries involving an arbitrary number of failures. For $f \geq 2$ we can build an $\tilde{O}(n^f)$-space data structure answering f-failure distance queries in $\tilde{O}(1)$ time. This compares favorably with the trivial $O(n^{f+2})$ space bound and the $\tilde{O}(n^{f+1})$ bound implied by [6, 3].

The High-Level Strategy. Our strategy for answering a dual-failure distance query is to systematically reduce it to queries that are qualitatively simpler or perhaps "smaller" in a certain sense. This brings up

*Email: {duanran, pettie}@umich.edu. This work was supported by NSF CAREER grant no. CCF-0746673.

[1]The notation $G - z$ refers to the graph G after removing z, where z is a vertex, edge, or set of vertices or edges.

the question of what makes for a *complicated* query. It turns out, perhaps contrary to intuition, that the most difficult queries to handle are when both failed vertices lie on the original shortest path; we call these *Case III* queries. Our objective is to reduce such a query to $O(1)$ *Case II* queries, where exactly one vertex lies on the original shortest path. We attempt to answer a Case II query by reducing it to $O(1)$ *Case I* queries, where the distance (measured in edges) from the source to one of the failed vertices is a power of 2. It may not always be possible to reduce a Case II query to Case I or to answer a Case I query directly. Many of our reductions are actually auto-reductions, where, for example, a Case II query is reduced to another Case II query that is measurably shorter. We can guarantee that the number of auto-reductions (i.e., the depth of the recursion) is only logarithmic. The high-level description of our query algorithm given above is not inaccurate but it does mask a truly intimidating number of sub-cases. Each of Cases I, II, and III has several varieties, each depending on the configuration of the two failures relative to some other critical vertices.

The moral conclusion we draw from our results is that handling dual-failure distance queries is possible but extending our data structure to handle 3 or more failures is practically infeasible. If there is a humanly comprehensible data structure for answering f-failure distance queries (for an arbitrary $f = O(1)$), it will probably not resemble our solution or its predecessors [6, 3].

Related Work. Our results fall into the genre of algorithms that *plan for failure*. In contrast to fully dynamic data structures, which allow the underlying graph to evolve in completely arbitrary ways, many algorithms [16, 4, 18, 12, 8, 7] implicitly operate under the assumption that the graph is essentially fixed and subject only to prescribed changes. Pătraşcu and Thorup [16] showed that an undirected graph could be preprocessed to answer connectivity queries after multiple failures. Specifically, given vertices s, t and edges $F = \{f_1, \ldots, f_d\}$, whether s and t are connected in $G - F$ can be determined in $\tilde{O}(d)$ time, where d is not a parameter of the preprocessing algorithm. Chan et al. [4] considered a model where the underlying graph is fixed but nodes can be flipped *on* or *off*. They showed that after $\tilde{O}(m^{4/3})$ preprocessing there is a data structure handling node updates (flips) in $\tilde{O}(m^{2/3})$ time and connectivity queries in $\tilde{O}(m^{1/3})$ time. Malik, Mittal, and Gupta [12] showed that for fixed nodes s, t in a weighted undirected graph, the distance $\delta_{G-e}(s, t)$ could be computed for *every* edge e in $O(m + n \log n)$ time. (This is often called the *replacement paths* problem.) This algorithm was rediscovered much later [8] in the context of a mechanism design problem [15], namely, computing the Vickrey prices for edges along a shortest path, where each edge is controlled by a single selfish agent. The time bound of the Malik et al. [12] algorithm was improved to $O(m\alpha(m, n))$ time for integer-weighted undirected graphs [13]. A similar $O(m + n \log n)$-time algorithm [14] was given for the vertex-failure problem, i.e., computing $\delta_{G-v}(s, t)$ for each vertex v. The replacement paths problem appears to be much more difficult on directed graphs. The trivial algorithm (see Yen [20] and Lawler [11]) takes $O(mn + n^2 \log n)$ time. For unweighted directed graphs, Roditty and Zwick [18] gave an $\tilde{O}(m\sqrt{n})$ time algorithm. Hershberger et al. [9] gave a lower bound of $\Omega(m\sqrt{n})$ for *weighted* directed graphs in the path-comparison model [10]. In recent work, Emek et al. [7] showed that when the underlying graph is *planar* the replacement paths problem (avoiding edges or vertices) is significantly easier to solve. Their algorithm runs in $O(n \log^3 n)$ time. The replacement paths problem is closely related to sensitivity analysis of shortest paths, i.e., to compute the amount by which each edge length can be changed until the identity of the shortest path changes. Pettie [17] showed that the sensitivity analysis of single source shortest paths (and minimum spanning trees) could be computed in $O(m \log \alpha(m, n))$ time.

2 Notations:

In this section we summarize the notation and conventions used throughout the paper.

- The query asks for the shortest path from x to y avoiding vertices u and v. We assume that at least one failed node, u, lies on the shortest path from x to y.

- We use $p_H(x, y)$ to denote the shortest path from x to y in the subgraph H and use xy as shorthand for $p_G(x, y)$, where G is the whole graph. The length and number of edges in a path p are denoted as $\|p\|$ and $|p|$, respectively. The concatenation of two paths p and p' is $p \cdot p'$. We use min to select the path with minimum length, i.e., $\min\{p_1, \ldots, p_k\}$ refers to the minimum length path among $\{p_1, \ldots, p_k\}$.

- We define the function $\rho_s(p_H(s, t))$ to be the vertex $c \in p_H(s, t)$ such that $|p_H(s, c)| = 2^{\lfloor \log |p_H(s,t)| \rfloor}$, i.e., $\rho_s(p_H(s, t))$ is the farthest vertex from s in the path $p_H(s, t)$, whose *unweighted* distance from s in H is a power of 2. Symmetrically, the function $\rho_t(p_H(s, t))$ is the vertex $c \in p_H(s, t)$ such that $|p_H(c, t)| = 2^{\lfloor \log |p_H(s,t)| \rfloor}$. It is easy to see $\rho_t(p_H(s, t))$ is *before* $\rho_s(p_H(s, t))$ in $p_H(s, t)$.

- Let $p_H(x,y) \diamond A$ be short for $p_{H\backslash A}(x,y)$. For example, our query is to determine $\|xy \diamond \{u,v\}\|$: the shortest x-y path avoiding u and v. Here A can be a range of vertices if the range is clear from context. For example, if we have established that s and t appear in $xy \diamond u$ then $xy \diamond u \diamond [s,t]$ refers to the shortest path from x to y avoiding u *and* the subpath from s to t within $xy \diamond u$.

- We let $s \oplus i$ and $s \ominus i$ be the ith vertex after s and *before* s on some path known from context. Typically the path we are considering is from x to y. For brevity we use $\oplus i$ and $\ominus i$ as short for $x \oplus i$ and $y \ominus i$, respectively. For example, $xy \diamond u \diamond (\ominus i)$ is the shortest path from x to y avoiding u and avoiding the ith vertex *before* y on the path $xy \diamond u$.

The following vertices are all *with respect to* some path $p_H(x,y)$ known from context, e.g., $p_H(x,y)$ may be $xy \diamond u$.

- Δ, ∇: The vertex at which $p_H(x,y)$ and xy first diverge is Δ, and, symmetrically, the first vertex where they converge is ∇.

- w, w': Let $w \in p_H(x,y)$ be the first vertex such that $wy = p_H(w,y)$; i.e., for *every* vertex *before* w, the shortest path to y goes through some vertex in $G\backslash H$. Symmetrically, $w' \in p_H(x,y)$ is the last vertex such that $xw' = p_H(x,w')$.

Throughout the paper we use the term *detour* to mean a (non-shortest) path avoiding some set of vertices.

3 Review of the One-Failure Distance Oracle

As in [6], throughout the paper we assume that all shortest paths are unique. Thus, we can determine if u is on *the* shortest path xy by checking whether $\|xu\| + \|uy\| = \|xy\|$.

Before delving into the description of our two-failure distance oracle, we first give a simplified version of the one-failure oracle [3, 6] that uses a log-factor more space: $O(n^2 \log^2 n)$.

3.1 Structure

- B_0: For every pair of vertices x and y, $B_0(x,y)$ stores the length and the number of vertices of xy. We also preprocess the shortest path trees [1] so that, given x_1, x_2, y, the first common vertex of x_1y and x_2y can be answered in constant time.

- B_1: For every pair of vertices x and y, $B_1(x,y)$ stores the length and number of vertices of the paths:

$$\begin{aligned} xy \diamond \{\oplus 2^i\} \quad &, \quad \forall i < \lfloor \log |xy| \rfloor \\ xy \diamond \{\ominus 2^i\} \quad &, \quad \forall i < \lfloor \log |xy| \rfloor \\ xy \diamond [\oplus 2^i, \ominus 2^j] \quad &, \quad \forall i,j < \lfloor \log |xy| \rfloor \end{aligned}$$

3.2 Query Algorithm Let the only failed vertex on the path xy be u. If $|xu|$ or $|uy|$ is an integer power of 2, then $xy \diamond u$ will be in B_1, so we can get the distance avoiding u immediately. Otherwise we will find $u_l = \rho_u(xu)$ and $u_r = \rho_u(uy)$ on xy, so $|u_l u|$ and $|uu_r|$ are powers of 2. There are 3 possible types of detours:

1. The detour that reaches some point in $(u, u_r]$.
2. The detour that reaches some point in $[u_l, u)$.
3. The detour that avoids the range $[u_l, u_r]$ in xy.

For the first and second types, the path will go through u_r or u_l. Since u is not on xu_l and $|u_l u|$ is a power of 2, xu_l is in $B_0(x, u_l)$ and $u_l y \diamond u$ is in $B_1(u_l, y)$. The concatenation of them is just the shortest path of the first type. In this situation, we say that the path $xu_l \cdot u_l y \diamond u$ **covers** this case. Symmetrically, $xu_r \diamond u \cdot u_r y$ can cover the second type. When we deal with the third type, we let $x' = \rho_x(xu)$ and $y' = \rho_y(uy)$, so $|xx'|$ and $|y'y|$ are integer powers of two and $xy \diamond [x', y'] \in B_1(x,y)$. Since x' is *after* u_l and y' is *before* u_r on xy, the detour avoiding $[u_l, u_r]$ must also avoid $[x', y']$, so $xy \diamond [x', y']$ covers the third type. (See Figure 1.) Thus, the single failure distance can be found in constant time.

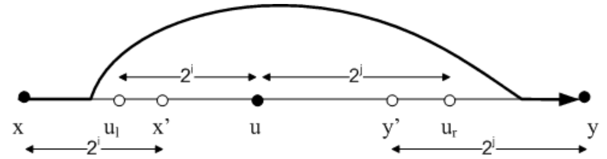

Figure 1: One-failure case, where the thick line denotes a detour of the third type.

In the following parts of this paper, we will consider the dual failure data structure in three cases:

- Section 4: Case I: only u is on xy and $|xu|$ or $|uy|$ is a power of 2.

- Section 5: Case II: only one failed vertex is on xy.

- Section 6: Case III: both u and v are on xy.

4 Case I

The first case we consider is when only one of the failed vertices u is on the original shortest path from x to y and $|xu|$ (or, symmetrically, $|uy|$) is a power of 2.

In Section 4.1 we present the data structures used in Case I. In Section 4.2 we present the query algorithm and dispense with several relatively easy subcases. Sections 4.2.1 and 4.2.2 cover the more complicated subcases of Case I.

4.1 Structures First we will introduce the data structures used in Case I, which are

4.1.1 Common Structures B_0, B_1: As described in the one-failure case.

B_2: For every detour $p_H(xy) \in B_1(x,y)$ and every $x' \in \{x, \Delta, w\}$, $y' \in \{y, \nabla, w'\}$ (x' is *before* y'), $B_2(x,y)$ stores the length and number of vertices of the paths:

$$\begin{aligned} p_H(xy) \diamond \{x' \oplus 2^i\}, &\quad i < \lfloor \log |p_H(xy)| \rfloor \\ p_H(xy) \diamond \{y' \ominus 2^i\}, &\quad i < \lfloor \log |p_H(xy)| \rfloor \\ p_H(xy) \diamond [x' \oplus 2^i, x' \oplus 2^{i+1}], &\quad i < \lfloor \log |p_H(xy)| - 1 \rfloor \\ p_H(xy) \diamond [y' \ominus 2^{j+1}, y' \ominus 2^j], &\quad j < \lfloor \log |p_H(xy)| - 1 \rfloor \end{aligned}$$

One can see the structures B_0, B_1, B_2 occupy $O(n^2 \log^3 n)$ space.

4.1.2 The Tree Structure In this section we introduce a specialized but useful data structure whose purpose will only become clear once it is seen in action, in Section 4.2. For every pair of vertices (u,y), define the sets $S(u,y)$ and $\hat{S}(u,y)$ as:

$$\begin{aligned} S(u,y) &= \{x \mid u \in xy \text{ and } |xu| \text{ is a power of } 2\} \\ \hat{S}(u,y) &= S(u,y) \cup \{z \mid \exists x_1, x_2 \in S(u,y) \\ &\quad s.t.\ z \text{ is the first common vertex of} \\ &\quad x_1 y \diamond u \text{ and } x_2 y \diamond u\} \end{aligned}$$

In the tree formed by the shortest paths from the vertex set of $S(u,y)$ to y in the subgraph $G-\{u\}$, $\hat{S}(u,y)$ is the set of all leaves and branch vertices in the tree, so $|\hat{S}(u,y)| \le 2|S(u,y)|$. Given a vertex y, every other vertex x can only be in at most $\log n$ different $S(u,y)$ since u must be on xy and $|xu|$ is a power of 2. Thus $\sum_u |S(u,y)| = n \log n$, and $\sum_{u,y} |S(u,y)| = n^2 \log n$.

For every pair of vertices (u,y) we store the following tree structure $T(u,y)$. For a given x let z^i be the 2^ith vertex of $\hat{S}(u,y)$ on the path $xy \diamond u$, i.e., $|(xz^i \diamond u) \cap \hat{S}(u,y)| = 2^i$. For each $x \in S(u,y)$ and i,j, we store in $T(u,y)$ the path $(xy \diamond u) \diamond [z^i, \ominus 2^j]$, where $\ominus 2^j$ is w.r.t. $xy \diamond u$. We also preprocess $T(u,y)$ to answer level ancestor and least common ancestor queries [1, 2] in the tree induced by $\hat{S}(u,y)$; this allows us to identify z^i and other vertices in $O(1)$ time. Obviously the size of $T(u,y)$ is $O(|\hat{S}(u,y)| \log^2 n)$, and the total space for the T structure is $O(n^2 \log^3 n)$.

Lemma 4.1. *Given $x_1, x_2 \in S(u,y)$ and an integer j, let z be the first common vertex of $x_1 y \diamond u$ and $x_2 y \diamond u$. Using the tree structure $T(u,y)$ we can find $\|(x_1 y \diamond u) \diamond [z, \ominus 2^j]\|$ in constant time.*

Proof. The vertex z can be identified in $O(1)$ time with a least common ancestor query. Let $i = \left\lfloor \log |(x_1 z \diamond u) \cap \hat{S}(u,y)| \right\rfloor$ be the log of the number of $\hat{S}(u,y)$-vertices on the path $x_1 z \diamond u$. Using two level ancestor queries we can identify z_l and x_1' where $|(z_l z \diamond u) \cap \hat{S}(u,y)| = 2^i$ and $|(x_1 x_1' \diamond u) \cap \hat{S}(u,y)| = 2^i$. The shortest detour $(x_1 y \diamond u) \diamond [z, \ominus 2^j]$ must be one of the following.

1. The detour that avoids the range $[z_l, \ominus 2^j]$ in $xy \diamond u$.
2. The detour that reaches some point in $[z_l, z)$.

Both of the paths $(x_1 y \diamond u) \diamond [x_1', \ominus 2^j]$ and $x_1 z_l \cdot (z_l y \diamond u) \diamond [z, \ominus 2^j]$ can be retrieved in $O(1)$ time from $T(u,y)$ and $B_0(x_1, z_l)$. These two paths cover both of the possibilities for $(x_1 y \diamond u) \diamond [z, \ominus 2^j]$. See Figure 2.

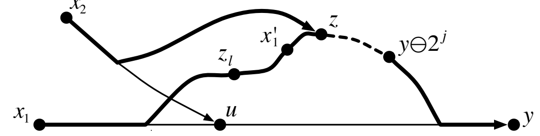

Figure 2: An illustration of the query $(x_1 y \diamond u) \diamond [z, \ominus 2^j]$, where we are given j, x_1, x_2, and y, but not z.

4.2 The detour from x to y avoiding u We begin with a simple observation:

Lemma 4.2. *Suppose for some distinct vertices u, v, x and y, v is on the detour $xy \diamond u$. Then at least one of u and v is on xy.*

Proof. Suppose u is not on xy, then $xy \diamond u = xy$, so $v \in xy$.

Since $|xu|$ is a power of 2, $xy \diamond u \in B_1(x,y)$. We determine if $v \in xy \diamond u$ by checking whether $\|xv \diamond u\| + \|vy \diamond u\| = \|xy \diamond u\|$ in constant time using the one-failure oracle. If $v \notin xy \diamond u$, the optimal detour has been found. If $|xv \diamond u|$ or $|vy \diamond u|$ is a power of 2, then we may return $(xy \diamond u) \diamond v$, which is in $B_2(x,y)$. Otherwise, we proceed to find $v_l = \rho_v(xv \diamond u)$ and $v_r = \rho_v(vy \diamond u)$ in $O(\log n)$ time as follows:

Since $|xu|$ is a power of 2, if $u \in xv$ then $xv \diamond u \in B_1(x,v)$, otherwise $xv \diamond u = xv \in B_0(x,v)$. Thus the

vertex v_l whose unweighted distance from v in $xv \diamond u$ is a power of 2 can be found in $B_2(x,v)$ or $B_1(x,v)$ in constant time.

However, since $|uy|$ is not necessarily a power of 2, v_r is not symmetrical to v_l. To locate v_r, we analyze how the path $vy \diamond u$ was constructed in the one-failure query algorithm. The only non-trivial case is when $vy \diamond u$ was composed of two parts (the first or second types, from Section 3.2), i.e., it was of the form $vu'_l \cdot u'_l y \diamond u$ or $vu'_r \diamond u \cdot u'_r y$, where $|uu'_r|$ and $|u'_l u|$ are powers of 2. We find some vertex v' that, depending on the form of $vy \diamond u$, is a maximal power of 2 from v, u'_l, or u'_r (in unweighted distance) but before v_r. We then continue to search for v_r on $v'y \diamond u$. Since $|v'y \diamond u| < |vy \diamond u|/2$ this procedure terminates after $O(\log n)$ steps. If v_r lies in vu'_l or $u'_r y$ then v' may be retrieved from B_1; if it lies in $vu'_r \diamond u$ or $u'_l y \diamond u$ then v' is stored in B_2.

The optimal detour avoiding u and v will belong to one of the following types:

1. The detour that avoids u and the range $[v_l, v_r]$ in $xy \diamond u$.

   The shortest detour of this kind must be no shorter than $(xy \diamond u) \diamond [\oplus 2^j, \oplus 2^{j+1}] \in B_2(x,y)$ or $(xy \diamond u) \diamond [\ominus 2^{j'+1}, \ominus 2^{j'}] \in B_2(x,y)$ where $j = \lfloor \log |xv \diamond u| \rfloor$ and $j' = \lfloor \log |vy \diamond u| \rfloor$. To see this, without loss of generality, assume $j < j'$. Then v_l is *before* $x \oplus 2^j$ and $|vv_r| = 2^{j'} > 2^j$, so v_r is *after* $x \oplus 2^{j+1}$ in $xy \diamond u$. Therefore, any detour avoiding $[v_l, v_r]$ belongs to the set of detours avoiding $[\oplus 2^j, \oplus 2^{j+1}]$ or $[\ominus 2^{j'+1}, \ominus 2^{j'}]$. (This is the same argument used in [6].)

2. The detour that reaches some points in $(v, v_r]$.

   In this case, the detour must go through v_r. Since $u, v \notin v_r y$, we just need to find the path $(xv_r \diamond u) \diamond v$. Suppose that $u \in xv_r$. Since $|xu|$ and $|vv_r \diamond u|$ are powers of 2, we can immediately return $(xv_r \diamond u) \diamond v \in B_2(x, v_r)$. Now suppose $u \notin xv_r$. Since $v \in xv_r \diamond u$, by Lemma 4.2 only v is on xv_r and $|vv_r|$ is a power of 2. Thus $xv_r \diamond v \in B_1(x, v_r)$. If $u \notin xv_r \diamond v$ (which can be checked with the one-failure oracle) we are done. If not, we return $(xv_r \diamond v) \diamond u$, which is stored in $B_2(x, v_r)$.

3. The detour that reaches some point in $[v_l, v)$, but does not reach $(v, v_r]$.

So now we only have to consider the last type of detour, which must go through v_l but not $(v, v_r]$. So far we have only ascertained that $v \in v_l y \diamond u$ and $|v_l v \diamond u|$ is a power of 2. From Lemma 4.2, at least one of u and v is on $v_l y$. We break the analysis into two main cases depending on whether v is in $v_l y$ (Case I.1) or not (Case II.2). In both cases, we begin by locating the u_l and u_r relative to u on the path $v_l y \diamond v$. We consider further subcases depending on whether $u_l \in xy \diamond u$:

- **I.1.a:** $v \in v_l y$ and $u_l \in xy \diamond u$
- **I.1.b:** $v \in v_l y$ and $u_l \notin xy \diamond u$
- **I.2.a:** $v \notin v_l y$ and $u_l \notin xy \diamond u$
- **I.2.b:** $v \notin v_l y$ and $u_l \in xy \diamond u$

4.2.1 Case I.1: v is on $v_l y$

Since v is not on uy, u cannot be *before* v on $v_l y$. So $u \notin v_l v$ and $|v_l v|$ is a power of 2. We check whether u is in $v_l y \diamond v$; if not, we are done. If $u \in v_l y \diamond v$, define $\overline{w}$ as the point w of the detour $v_l y \diamond v$, i.e., the first vertex in $v_l y \diamond v$ which satisfies $v \notin \overline{w} y$. Since $v \notin uy$, u must be equal to or *after* $\overline{w}$ on $v_l y \diamond v$. If $u = \overline{w}$, then $(v_l y \diamond v) \diamond \overline{w}$ is in $B_2(v_l, y)$. Otherwise we can find $u_l = \rho_u(\overline{w} u \diamond v)$ and $u_r = \rho_u(uy \diamond v)$ in $O(\log n)$ time as in Section 4.2. So u_l is *after* $\overline{w}$ on $v_l y \diamond v$, and $v \notin u_l y$. The possible types of detours from v_l to y are:

1. The detour that avoids v and the range $[u_l, u_r]$ in $v_l y \diamond v$.

2. The detour that reaches some point in $(u, u_r]$.

3. The detour that reaches some point in $[u_l, u)$, but does not reach $(u, u_r]$.

Similar to the discussion in Section 4.2, the first case can be covered by the paths $(v_l y \diamond v) \diamond [\overline{w} \oplus 2^k, \overline{w} \oplus 2^{k+1}], (v_l y \diamond v) \diamond [\ominus 2^{k'+1}, \ominus 2^{k'}] \in B_2(v_l, y)$ for some k, k', and the second case can also be handled in the same way. Thus we only have to consider the third case, that is, the path from u_l to y avoiding u and $(v, v_r]$. Since $v \notin u_l y$ and $|u_l u|$ is a power of 2, $u_l y \diamond u$ is in $B_1(u_l, y)$. We check whether u_l is on $xy \diamond u$ in constant time and have the following subcases:

Case I.1.a When $u_l \in xy \diamond u$, we know that $v_l \in xy \diamond u$ and $u \notin v_l v$. Furthermore, u_l is not in the range $[v_l, v)$ on $xy \diamond u$. To see this, assume u_l is on $v_l v \diamond u = v_l v$ (the range $[v_l, v)$ on $xy \diamond u$). Since $v \in v_l y$, u_l is *before* v on $v_l y$ and $v \in u_l y$, which contradicts the fact that $v \notin u_l y$.

If u_l is *after* v on $xy \diamond u$, then $v \notin u_l y \diamond u$, so we just return $u_l y \diamond u$. We do not need to consider the case when u_l is *before* v_l on $xy \diamond u$ since any detour that goes through v_l then u_l contains a cycle.

Case I.1.b When $u_l \notin xy \diamond u$, since $u \in xy, u \in u_l y$ and $|xu|, |u_l u|$ are both powers of 2, x and u_l are both in $S(u, y)$. From Lemma 4.1, we can find the least common ancester z of x and u_l in $T(u, y)$ in constant time [19], i.e., z is the first common vertex of the shortest paths $xy \diamond u$ and $u_l y \diamond u$. (See Figure 3.) If $v \notin u_l y \diamond u$, just

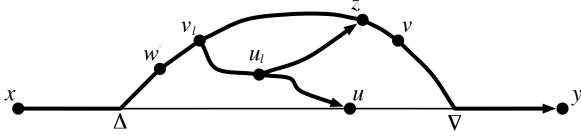

Figure 3: The usage of tree structure in Case I.1.b.

return $u_l y \diamond u$. If $v \in u_l y \diamond u$, v must be *after* z in the path $u_l y \diamond u$ because $v \in xy \diamond u$.

Assume the shortest detour reaches u_l then to some point in the common range $[z, v)$ of $xy \diamond u$ and $u_l y \diamond u$. Since $u_l \notin xy \diamond u$, the path from x through $xy \diamond u$ to $[z, v)$ must be shorter than that detour. Thus we do not need to consider the detours that pass through u_l then to some vertices in $[z, v)$ of $u_l y \diamond u$.

Since we are in the third type of Section 4.2, in which the range $(v, v_r]$ is avoided, we just have to find $(u_l y \diamond u) \diamond [z, v_r]$, which can be covered by $(u_l y \diamond u) \diamond [z, \ominus 2^{j'}]$ (where $j' = \lfloor \log |vy \diamond u| \rfloor$) since $v_r = v \oplus 2^{j'}$ is *after* $y \ominus 2^{j'}$ on $xy \diamond u$. By Lemma 4.1, this can be achieved in constant time using the $T(u, y)$ structure.

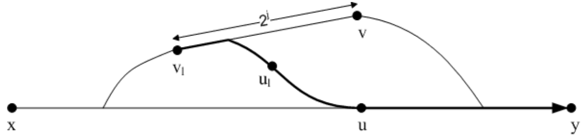

Figure 4: The illustration of the case that u_l is not on $xy \diamond u$

4.2.2 Case I.2: v is not on $v_l y$

Since v is on $v_l y \diamond u$, by Lemma 4.2, u must be on $v_l y$. We find the vertices $u_l = \rho_u(v_l u)$ and $u_r = \rho_u(uy)$. There are two further possible cases:

Case I.2.a If u_l is not on $xy \diamond u$, this case is very similar to Case I.1.b. The three possible types of detours are:

1. The detour that avoids the range $[u_l, u_r]$ in $v_l y$ and the vertex v.
2. The detour that reaches some point in $(u, u_r]$.
3. The detour that reaches some point in $[u_l, u)$, but does not reach $(u, u_r]$.

The first type is clearly in $B_2(v_l, y)$, since $(v_l y \diamond [u_l, u_r]) \diamond v$ can be covered by $v_l y \diamond [\oplus 2^j, \ominus 2^{j'}] \diamond v$ ($j = \lfloor \log |v_l u| \rfloor$ and $j' = \lfloor \log |uy| \rfloor$), and the number of vertices between v_l and v in that path is a power of 2 (see Figure 4). The second type is also similar to Section 4.2. But for the third type, the detour must reach u_l, so we have to find the path $u_l y$ avoiding u and v. Since u_l and x are both in $S(u, y)$, by the same argument of Case I.1.b, we only need to find the path $(u_l y \diamond u) \diamond [z, v_r]$, where z is the first common vertex of $xy \diamond u$ and $u_l y \diamond u$. By utilizing the tree structure $T(u, y)$, the path $(u_l y \diamond u) \diamond [z, \ominus 2^{j'}]$ where $j' = \lfloor \log |vy \diamond u| \rfloor$ can be answered in constant time.

Case I.2.b If u_l is on $xy \diamond u$, there are two kinds of detours since our overall goal is to find $(xy \diamond u) \diamond (v, v_r]$:

1. The detour $(xy \diamond u) \diamond [u_l, v_r]$
2. The detour that reaches some point in $[u_l, v)$

For the first kind, it is easy to see both x and u_l are in $S(u, y)$, so they are in the tree structure $T(u, y)$. We can find the detour $(xy \diamond u) \diamond [u_l, \ominus 2^{j'}]$ where $j' = \lfloor \log |vy \diamond u| \rfloor$ in constant time, which will cover the first kind.

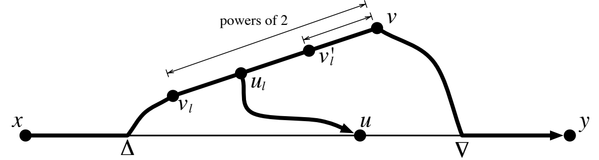

Figure 5: The illustration of Case I.2.a.iii, where v_l' is the corresponding v_l for the path $u_l y \diamond u$.

For the second kind, the detour will reach u_l through $xu_l \diamond u$. Since only u is on $u_l y$ and $|u_l u|$ is a power of 2, $u_l y \diamond \{u, v\}$ itself is in Case I, and we can deal with it recursively by the procedure of Case I. When we try to find the detour from u_l to y avoiding u and v by the procedure in Section 4.2, the position of v_r has not changed, so we do not need another $O(\log n)$ time to locate it. Furthermore the new v_l' found must be such that $|v_l' v \diamond u|$ is a *smaller* power of 2 than $|v_l v \diamond u|$; see Figure 5. Thus, the number of recursive invocations of Case I is $O(\log n)$.

5 Case II: One failed vertex on xy

In Case II we deal with the situation where only one failed vertex is on xy. Our strategy is to systematically reduce such a query to several Case I queries. A full description of Case II will appear in the full version of this paper.

6 Case III: Two failed vertices on xy

In this case both u and v are on the original shortest path from x to y, where u is *before* v in xy. In Section 6.1 we consider the situation where $|xu|$ or $|vy|$ is a power of 2; these queries are easily reducible to several Case I queries. However, in general we will need to use a fundamentally different approach to answering such queries. In Section 6.2 we introduce a *binary partition* data structure that is tailored to Case III queries and in Section 6.3 we give the complete Case III query algorithm.

6.1 If $|xu|$ or $|vy|$ is a power of 2 W.l.o.g, we only consider the case where $|xu|$ is a power of 2 and $v \in xy \diamond u$. As in Section 4.2, we find $v_l = \rho_v(\nabla v)$ and $v_r = \rho_v(vy)$, where ∇ is the convergence point of the paths $xy \diamond u$ and xy. The shortest detour belongs to one of the following types:

1. The detour that avoids u and the range $[v_l, v_r]$ in $xy \diamond u$.
2. The detour that reaches some points in $(v, v_r]$.
3. The detour that reaches some point in $[v_l, v)$, but does not reach $(v, v_r]$.

The first and second types are the same as in Section 4.2, and the third type is reducible to Case I since $|v_l v|$ is a power of 2.

6.2 The binary partition structure When both failed vertices lie on the shortest path xy we need to consider the possibility that the optimal detour departs from xy before u and returns to xy between u and v, possibly departing and returning several times. If we could identify with certainty just *one* vertex m between u and v that lies on $xy \diamond \{u, v\}$, we could reduce our Case III query to two Case II queries: $xm \diamond \{u, v\}$ and $my \diamond \{u, v\}$. The binary partition structure allows us to answer a Case III query directly or reduce it to Case II queries. For each x, y and $i, j \le \lfloor \log |xy| \rfloor$, we store the following structure $C_{i,j}(x, y)$:

Let $[x', y'] = [x \oplus 2^i, y \ominus 2^j]$. Define the following points on $[x', y']$:

$$m_{q,r} \in x'y', \qquad \text{such that } |x'm_{q,r}| = \left\lfloor \frac{r}{2^q} |x'y'| \right\rfloor, \quad \text{for all } 1 \le q \le \lceil \log |x'y'| \rceil, \ \ 0 \le r \le 2^q$$

These points define the following ranges:

$$R_{q,0} = [m_{q,0}, m_{q,1}], \forall 1 \le q \le \lceil \log |x'y'| \rceil$$

and

$$R_{q,r} = (m_{q,r}, m_{q,r+1}], \forall 1 \le q \le \lceil \log |x'y'| \rceil, 1 \le r \le 2^q - 1$$

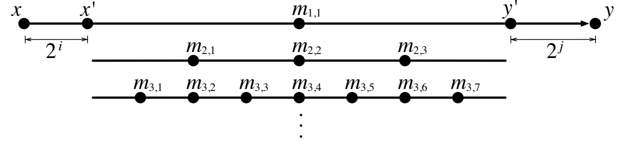

Figure 6: Different levels of the binary structure

where $R_{q,2^q-1}$ is truncated at y'. See Figure 6.

Thus, in level q, we have 2^q disjoint ranges $R_{q,0}, R_{q,1}, \ldots, R_{q,2^q-1}$, and their union is the whole range $[x', y']$. For every level q, we store in $C_{i,j}(x, y)$ the length and number of vertices of the following paths. (Below the superscripts are mnemonics, where e, o, l, f, and b are for even, odd, last, first, and backwards.) The space needed for this structure is $O(n^2 \log^3 n)$.

1. p^e_q:
$$p^e_q = \min_{\substack{r \in [0, 2^q) \\ r \text{ even}}} xy \diamond (x'y' \setminus R_{q,r})$$
Let r^e_q be the index r for p^e_q. That is, among all paths from x to y that intersect *only one* of the even intervals, we store the one with minimum length. Define l^e_q to be the leftmost vertex of p^e_q in the range R_{q,r^e_q}, that is, $l^e_q \in p^e_q \cap R_{q,r^e_q}$ that minimizes $|xl^e_q|$.

2. p^o_q:
$$p^o_q = \min_{\substack{r \in [0, 2^q) \\ r \text{ odd}}} xy \diamond (x'y' \setminus R_{q,r})$$
Let r^o_q be the index r for p^o_q. Store s^o_q as the rightmost vertex of p^o_q in the range R_{q,r^o_q}.

3. p^{el}_q: Define the last vertex on p^e_q which is in the subrange R_{q,r^e_q} as L^e_q. Store the path:
$$p^{el}_q = xy \diamond (x'y' \setminus (L^e_q, m_{q,r^e_q+1}])$$
i.e., p^{el}_q may only use vertices in the range $(L^e_q, m_{q,r^e_q+1}]$.

4. p^{of}_q: Define the first vertex on p^o_q which reaches the subrange R_{q,r^o_q} as F^o_q. Store the path:
$$p^{of}_q = xy \diamond (x'y' \setminus (m_{q,r^o_q}, F^o_q))$$

Parts 5-9 will use the following notation:

Let S and T be two disjoint adjacent subpaths in $x'y'$, where S precedes T, and let $X = xx', Y = y'y$. Let X' be the subpath between X and S and Y' be the subpath between T and Y. See Figure 7.

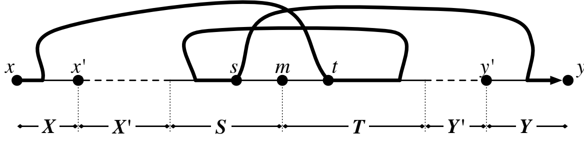

Figure 7: The form of the path $D(S,T)$. Here S,T are arbitrary adjacent intervals on xy and $X = xx'$ and $Y = y'y$.

Obviously X, X', S, T, Y', Y are disjoint and form the path xy. Define the path $D(S,T)$ to be:

$$D(S,T) = \min_{s\in S, t\in T}(xt \diamond (X' \cup S)) \cdot (ts \diamond (st \setminus \{t,s\})) \cdot (sy \diamond (T \cup Y'))$$

That is, $D(S,T)$ is the shortest path from x to y that passes through T then S, and that never returns to T and avoids all other vertices in $x'y'$.

5. p_q^b:

$$p_q^b = \min_{\substack{r\in[0,2^q-2]\\ r \text{ even}}} D(R_{q,r}, R_{q,r+1})$$

Let r_q^b denote the index r for p_q^b. Store l_q^b and s_q^b as the leftmost and rightmost vertex of p_q^b in the range $R_{q,r_q^b} \cup R_{q,r_q^b+1}$.

6. p_q^{bf}: Define the first vertex on p_q^b which reaches the subrange R_{q,r_q^b+1} as F_q^b, and store

$$p_q^{bf} = D(R_{q,r_q^b}, (m_{q,r_q^b+1}, F_q^b))$$

I.e., it further avoids the range $[F_q^b, m_{q,r_q^b+2}]$ from p_q^b. Store l_q^{bf} as the leftmost vertex of p_q^{bf} in the range R_{q,r_q^b}.

7. p_q^{bfl}: Let the last vertex on p_q^{bf} in the subrange R_{q,r_q^b} be L_q^{bf} and store the path:

$$p_q^{bfl} = D((L_q^{bf}, m_{q,r_q^b+1}], (m_{q,r_q^b+1}, F_q^b))$$

Figure 9 in Section 6 illustrates this path.

8. p_q^{bl}: Let the last vertex on p_q^b in the subrange R_{q,r_q^b} be L_q^b and store the path:

$$p_q^{bl} = D((L_q^b, m_{q,r_q^b+1}], R_{q,r_q^b+1})$$

Store s_q^{bl} as the rightmost vertex of p_q^{bl} in the range R_{q,r_q^b+1}.

9. p_q^{blf}: Define the first vertex on p_q^{bl} which is in the subrange R_{q,r_q^b+1} to be F_q^{bl}, and store:

$$p_q^{blf} = D((L_q^b, m_{q,r_q^b+1}], (m_{q,r_q^b+1}, F_q^{bl})).$$

6.3 General Cases We find $u_l = \rho_u(xu)$ and $v_r = \rho_v(vy)$ in constant time. The optimal detour can belong to one (or more) of the following types:

- III.1 The detour that reaches some point in $(v, v_r]$.
- III.2 The detour that reaches some point in $[u_l, u)$.
- III.3 The detour that avoids $[u_l, v_r]$
- III.4 The detour that avoids $[u_l, u]$ and $[v, v_r]$ in xy, but reaches some vertex between (u, v).

The first and second are considered in Section 6.1. The third one can also be covered by finding $x' = \rho_x(xu)$ and $y' = \rho_y(vy)$ and then returning $xy \diamond [x', y'] \in B_1(x,y)$. However, things become more complicated when we consider the fourth case, which means the detour leaves xy *before* u_l and merges with xy *after* v_r and goes through some vertex between u and v. To deal with this case, we will need the binary partition structure introduced in the previous subsection.

Now consider the positions of u and v. Find the smallest level q in $C_{i,j}(x,y)$ ($i = \log|xx'|$, $j = \log|y'y|$) in which u and v are not in the same subrange. (This can be achieved by computing $|xu|$ and $|xv|$.) Let $u \in R_{q,r}$ and $v \in R_{q,r+1}$, where r is even. (If r is odd, then u and v are also in different subranges in level $q-1$.) Denote the rightmost vertex of $R_{q,r}$ by m. There are 4 possible types for detour *III.4*:

- III.4.a The shortest detour only goes through the vertices in $R_{q,r}$.
- III.4.b The shortest detour only goes through the vertices in $R_{q,r+1}$,
- III.4.c The shortest detour goes through some vertices in $R_{q,r}$, then to some vertices in $R_{q,r+1}$.
- III.4.d The shortest detour goes through some vertices in $R_{q,r+1}$, then to some vertices in $R_{q,r}$ but does not reach m.

In Case III.4.a there are some possible subcases depending on the relative positions of u and the path p_q^e. See Figure 8.

- III.4.a.i If p_q^e does not go through $R_{q,r}$ in $C_{i,j}(x,y)$, then there exists another path that only goes through R_{q,r_q^e} disjoint to $R_{q,r}$ but shorter than any

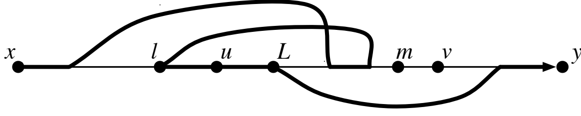

Figure 8: The illustration of the positions of u, L and m.

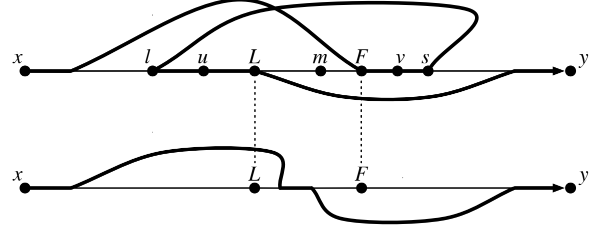

Figure 9: The fourth type in Case III

path only going through $R_{q,r}$. So p_q^e goes through some vertices in $[x', y']$ but does not touch the range $[u, v]$ in xy. Thus, it has already been covered by Cases III.1 or III.2, as we discussed above.

- III.4.a.ii If L_q^e is *before* u in xy, then p_q^e must be longer than $\|xL_q^e\| + \|L_q^e y \diamond [u, v]\|$, which will go through u_l. This possibility was dealt with in Case III.2.

- III.4.a.iii If u is *before* l_q^e, p_q^e is the shortest detour for Case III.4.a. (Remember here l_q^e is the leftmost vertex of p_q^e in the range R_{q,r_q^e}.)

- III.4.a.iv If $u \in [l_q^e, L_q^e]$, there are two types of detours depending on whether it goes through the range $(u, L_q^e]$. From the definition of p_q^e, a shortest path that travels through some vertices in $(u, L_q^e]$ must travel through L_q^e. Thus, $xy \diamond \{u, v\}$ will be the concatenation of the paths from x to L_q^e and from L_q^e to y avoiding u and v, which are both in Case II. For the detours not going through the range $(u, L_q^e]$, p_q^{el} can cover this case.

The Case III.4.b is symmetric to Case III.4.a: just replace p_q^e by p_q^o, L_q^e by F_q^o, l_q^e by s_q^o, and $R_{q,r}$ by $R_{q,r+1}$.

For the Case III.4.c, the shortest detour must go through the vertex m which separates these two ranges $R_{q,r}$ and $R_{q,r+1}$. Just find the paths from x to m and from m to y avoiding u and v, which are both in Case II.

For Case III.4.d, there are some possible subcases depending on the relative positions of u, v and the path p_q^b. See Figure 9:

- III.4.d.i If p_q^b does not go through $R_{q,r}$ or $R_{q,r+1}$, i.e., $r \neq r_q^b$, then the shortest detour has already been covered by Cases III.1 or III.2

- III.4.d.ii If L_q^b is *before* u or F_q^b is *after* v in xy, we have already considered this situation in Cases III.1 and III.2.

- III.4.d.iii If $u \in [l_q^b, L_q^b]$, then any detours that reach some vertex in $(u, L_q^b]$ will go through L_q^b. To cover the possibility that the shortest detour goes through some vertices in $(u, L_q^b]$, we find the detours from x to L_q^b and from L_q^b to y avoiding u and v, which are both in Case II. To cover the possibility that the shortest detour avoids $(u, L_q^b]$, we can see p_q^{bl} satisfies this condition. Then in the path p_q^{bl}, there are some subcases:
  - If v is *after* s_q^{bl} in xy, p_q^{bl} is the shortest detour for this case.
  - If F_q^{bl} is *after* v in xy, p_q^{bl} must be longer then $\|xF_q^{bl} \diamond [u, v]\| + \|F_q^{bl} y\|$, which will go through v_r. This situation has been covered by Case III.2.
  - If $v \in [F_q^{bl}, s_q^{bl}]$, then any detours which reach some vertex in $[F_q^{bl}, v)$ will go through F_q^{bl}, so it can be covered by $xF_q^{bl} \diamond \{u, v\} \cdot F_q^{bl} y \diamond \{u, v\}$, which are both in Case II. Furthermore we can use the path p_q^{blf} to cover the case that it does not go through F_q^{bl}.

- III.4.d.iv If $v \in [F_q^b, s_q^b]$, it is symmetric to the Case III.4.c.iii.

- III.4.d.v If u is *before* l_q^b and v is *after* s_q^b, then p_q^b is just the shortest detour for III.4.d.

This concludes the query algorithm for Case III. The total running time will be $O(\log n)$, which comes from the auto-reductions in Case I.

7 Conclusion

We have shown that dual-failure distance queries can be answered in near-constant time with a data structure of nearly minimal space. The complexity of our solution suggests that any feasible distance oracle for the f-failure problem (for arbitrary $f = O(1)$) should be different than ours in some fundamental way. A promising

avenue for further research is to consider weaker or more restrictive versions of the problem. For example, little is known about the complexity of the problem on unweighted undirected graphs, or in unweighted directed graphs when only connectivity queries are supported.

References

[1] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings 4th Latin American Symp. on Theoretical Informatics (LATIN), LNCS Vol. 1776*, pages 88–94, 2000.

[2] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[3] A. Bernstein and D. Karger. Improved distance sensitivity oracles via random sampling. In *Proceedings 19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 34–43, 2008.

[4] T. M. Chan, M. Pătraşcu, and L. Roditty. Dynamic connectivity: Connecting to networks and geometry. In *Proceedings 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2008.

[5] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[6] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. Comput.*, 37(5):1299–1318, 2008.

[7] Y. Emek, D. Peleg, and L. Roditty. A near-linear time algorithm for computing replacement paths in planar directed graphs. In *Proceedings 19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 428–435, 2008.

[8] J. Hershberger and S. Suri. Vickrey prices and shortest paths: what is an edge worth? In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 252–259, 2001. Erratum, Proc. 43rd FOCS, p. 809, 2002.

[9] J. Hershberger, S. Suri, and A. Bhosle. On the difficulty of some shortest path problems. In *Proceedings 20th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 343–354, 2003.

[10] D. R. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Comput.*, 22(6):1199–1217, 1993.

[11] E. L. Lawler. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem. *Management Sci.*, 18:401–405, 1971/72.

[12] K. Malik, A. K. Mittal, and S. K. Gupta. The k most vital arcs in the shortest path problem. *Oper. Res. Lett.*, 8(4):223–227, 1989.

[13] E. Nardelli, G. Proietti, and P. Widmayer. A faster computation of the most vital edge of a shortest path. *Info. Proc. Lett.*, 79(2):81–85, 2001.

[14] E. Nardelli, G. Proietti, and P. Widmayer. Finding the most vital node of a shortest path. *Theoretical Computer Science*, 296(1):167–177, 2003.

[15] N. Nisan and A. Ronen. Algorithmic mechanism design. *Games Econom. Behav.*, 35(1–2):166–196, 2001. Economics and artificial intelligence.

[16] M. Pătraşcu and M. Thorup. Planning for fast connectivity updates. In *Proceedings 48th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 263–271, 2007.

[17] S. Pettie. Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time. In *Proceedings 16th Int'l Symposium on Algorithms and Computation (ISAAC)*, pages 964–973, 2005.

[18] L. Roditty and U. Zwick. Replacement paths and k-simple shortest paths in unweighted directed graphs. In *Proceedings 32nd Int'l Colloq. on Automata, Languages, and Programming (ICALP)*, pages 249–260, 2005.

[19] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[20] J. Y. Yen. Finding the K shortest loopless paths in a network. *Management Sci.*, 17:712–716, 1970/71.