Oracles for Distances Avoiding a Link-failure

Camil Demetrescu *
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Rome, Italy

Mikkel Thorup †
AT&T Labs-Research
180 Park Avenue
Florham Park, NJ 07932

Abstract

For a directed graph G we consider queries of the form: "What is the shortest path distance from vertex x to vertex y in G avoiding a failed link (u, v), and what edge leaving x should we use to get on a such a shortest path?" We show that an oracle for such queries can be stored in $O(n^2 \log n)$ space with a query time of $O(\log n)$. No non-trivial solution was known for this problem.

1 Introduction

In the *Distance Sensitivity* problem we wish to preprocess a data structure (*Distance Sensitivity Oracle*) for a weighted directed graph $G = (V, E)$ that supports any sequence of the following query operations:

`distance(x,y,u,v)`: return the shortest distance from vertex x to vertex y in G using paths that do not contain edge (u, v);

`path(x,y,u,v)`: return a shortest path from vertex x to vertex y in G, if any, which does not contain edge (u, v).

A variant of this problem, related to reachability in directed acyclic graphs, has been first introduced by King and Sagert in [5], where they consider the problem of supporting *sensitivity queries* of the kind: "Is there a path from vertex x to vertex y which does not contain edge (u, v)?" Here, we are concerned with general weighted digraphs and distance queries instead of reachability queries.

Our motivating scenario is a network where link-failures happen quite rarely. As soon as a link-failure has been noticed, we want to be able to answer distance queries and provide directions for shortest paths in the graph without the failed link. On the other hand, we assume that we have plenty of time to compute a new data structure in the back-ground.

*Email: demetres@dis.uniroma1.it. URL: http://www.dis.uniroma1.it/~demetres. Author supported in part by the DIMACS Special Focus on Next Generation Networks Technologies and Applications, by the IST Programme of the EU under contract n. IST-1999-14.186 (ALCOM-FT) and by the Italian Ministry of University and Scientific Research (Project "Algorithms for Large Data Sets: Science and Engineering". This work has been done while visiting the AT&T Shannon Labs, 180 Park Avenue, Florham Park, NJ 07932.

†Email: mthorup@research.att.com. URL: http://www.research.att.com/info/mthorup.

The most natural approach to this problem would be to use a dynamic all pairs shortest path (APSP) algorithm, and delete the failed link (u, v). However, at present, the best update bound for maintaining explicitly APSP in directed graphs with real edge weights is $O(Sn^{2.5}\log^3 n)$ amortized time, where S is the maximum number of different real weights that each edge can assume throughout the sequence of updates [1]. Even in the special case of integer edge weights less than C, maintaining APSP requires $O(n^{2.5}\sqrt{C\log n})$ amortized time per update [4]. Thus, if the goal is to answer queries as quickly as possible, this might be better than just answering each query with a single source shortest path (SSSP) computation taking, e.g., $O(n\log n + m)$ time using Fredman and Tarjan's Fibonacci heaps [3].

However, our main target here is to answer queries quickly just after a link-failure. The most obvious solution would be to make a table of size $O(n^2 m)$ that for each vertex pair (x, y) and each edge (u, v) stored the distance from x to y avoiding (u, v). Such a table is trivially computed by m APSP computations. For sparse graphs, this takes $\tilde{O}(m^2 n)$ time[1], and for dense graphs, it takes $O(mC^{0.681}n^{2.575})$ [7].

A simple improvement in both time and space is that we work from one source s at the time, constructing a shortest path tree $T(x)$. For each vertex y, we know that the distance is only affected if (u, v) is one of the at most $n-1$ edges between x and y in T. Hence it is only for these edges that we need to record new distances to y. As a result, the total space is improved from $O(mn^2)$ to $O(n^3)$. The query time is still $O(1)$. As elaborated in [6], it is also quite easy to get the running time down from $\tilde{O}(m^2 n)$ to $\tilde{O}(mn^2)$. Still, we consider the cubic space prohibitive.

Our Results. As a main contribution, we show that, surprisingly, it is possible to design a deterministic oracle with fast query time that uses nearly the same space required for storing a distance matrix. More precisely, we construct an oracle that uses $O(n^2\log n)$ space and answers distance queries with a link-failure in $O(\log n)$ worst-case time.

The construction time for our oracle is $O(mn^2\log n + n^3\log^2 n)$ worst-case time. However, if one is willing to settle for $O(n^{2.5})$ space, we can improve preprocessing time to $O(mn^{1.5} + n^{2.5}\log n)$, and then the query time becomes constant.

Other applications. One application of our result is that in $\tilde{O}(m)$ time we can compute the consequences of an arbitrary weight change, like a one-move dynamic graph algorithm. More precisely if the weight of (u, v) is changed, we compute all distances to u and all distances from v with two SSSP computations. Now, to find the distance from x to y via (u, v), we just add the distance from x to u, the new weight of (u, v), and the distance from v to y. We can then compare this with the distance from x to v with (u, v) failed.

The ability to deal with single weight changes is interesting in connection with a local search like the one in [2]. There, one wants to consider a neighborhood of a given weight setting w, where each neighbor is obtained by changing a single weight.

Techniques. To achieve our bounds, we preprocess data structures where we store information about all pairs shortest paths obtained by excluding only edges with specific properties, rather than excluding all possible edges. We also store information about shortest paths where we exclude entire paths, rather than single edges. Space is compressed to $O(kn^2)$, with $k = \log n$ and $k = \sqrt{n}$ for the two oracles, since the number of excluded

[1] Throughout the paper, we use $\tilde{O}(f(n))$ to denote $O(f(n)\mathrm{polylog}(n))$.

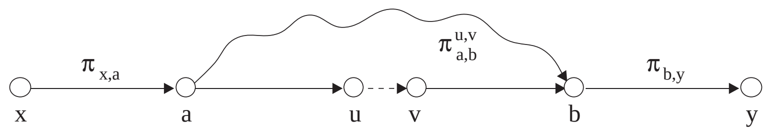


Figure 1: Optimal detour $\pi^{u,v}_{a,b}$ for vertices x, y in G obtained by excluding edge (u, v).

edges and paths never exceeds $O(k)$ for each pair x, y. While excluding single edges is relatively easy, a nice combinatorial property on trees helps us compute quite efficiently shortest paths that do not use entire paths. To answer queries, we show that shortest paths that do not contain any given edge can always be obtained as concatenation of at most two subpaths for which the answer is stored explicitly in the data structure.

Notation. Throughout the paper, we denote by $T(x)$ a shortest paths tree of G rooted at vertex x and by $\pi_{x,y}$ the unique path from vertex x to vertex y in $T(x)$. Without loss of generality, we assume that, if $z \in \pi_{x,y}$, then $\pi_{x,z}, \pi_{z,y} \subseteq \pi_{x,y}$. We also denote by $w_{x,y}$ the weight of edge (x, y) in G, by $d_{x,y}$ the weighted distance from x to y in G, and by $h_{x,y}$ the number of edges in $\pi_{x,y}$. By $\widehat{G}$ we denote graph G where we have reversed the orientation of edges. Thus, $\widehat{\pi}$ and $\widehat{T}$ denote π and T related to $\widehat{G}$. Moreover, we let $B_x(i, j)$ denote the band of paths in $T(x)$ that connect vertices at level i with vertices at level $j > i$. Notice that, if $\pi_{c,d} \in B_x(i, j)$ then $h_{c,d} = j - i$. Finally, $\pi^{u,v}_{x,y}$ denotes a shortest path from vertex x to vertex y in G obtained by excluding edge (u, v). Let a and b be any vertices in $\pi_{x,y}$ such that

$$\pi^{u,v}_{x,y} = \pi_{x,a} \cdot \pi^{u,v}_{a,b} \cdot \pi_{b,y} \quad \text{and} \quad \pi_{a,b} \cap \pi^{u,v}_{a,b} = \emptyset$$

where operator $\cdot$ denotes path concatenation (see Figure 1). Observe that $\pi^{u,v}_{a,b}$ is the "best detour" that we can follow if we wish to go from vertex x to vertex y assuming to exclude edge (u, v), and that vertices a and b mark the longest common portions of $\pi_{x,y}$ and $\pi^{u,v}_{x,y}$.

2 Computing Shortest Distances by Excluding Paths

In this section we discuss simple algorithms for computing shortest distances by excluding paths in a directed graph G with nonnegative real-valued edge weights. These algorithms will be central in Section 3 and Section 4 to preprocessing distance sensitivity oracles. In particular, we consider the problem of designing a procedure $\texttt{exclude}(G, T(x), i, j)$ that computes the shortest distances from vertex x to all vertices $y \in T(x)$ at level $\geq j$, obtained by excluding from G the edges on the portion of $\pi_{x,y}$ that falls in $B_x(i, j)$.

We first note that, since $|B_x(i, j)| \leq O(n)$, $\texttt{exclude}(G, T(x), i, j)$ can be easily implemented in $O(mn + n^2 \log n)$ time by running Dijkstra's algorithm [3] for each excluded path in $B_x(i, j)$. However, if $B_x(i, j)$ contains only edge-disjoint paths, then we can devise a much faster version of `exclude`, denoted by `exclude-d`, which requires $O(m + n \log n)$ worst-case time.

Algorithm `exclude-d`. We assume that paths in $B_x(i, j)$ are edge-disjoint. This implies that for any vertex u at level $i + 1$ there is at most a descendent of u at level j, which we denote by $\bar{u}$. In our description, we are only interested in vertices at level $i + 1$ that have a descendent at level j. The main idea behind our algorithm is that, if v and w are any vertices at level $i + 1$ in $T(x)$, then the shortest paths from vertex x to any vertex

y in the subtree of $T(x)$ rooted at v are not affected by removing edges in $\pi_{w,\bar{w}}$. As a consequence, we can work independently on the vertices of each subtree of $T(x)$ rooted at level $i+1$. The algorithm `exclude-d`$(G, T(x), i, j)$ works as follows. For each vertex u at level $i+1$ we compute a shortest paths tree rooted at vertex x of the weighted directed graph $G_u = (V_u, E_u)$, where:

- V_u contains vertex x, vertex u, and all the descendents of u in $T(x)$;
- E_u contains edges going from x to any other vertex in V_u and contains edges in G induced by vertices in V_u except for the edges in $\pi_{u,\bar{u}}$;
- $w_{u,a,b}$ is the weight of edge (a, b) in G_u defined as:

$$w_{u,a,b} = \begin{cases} \min_{c \notin T_x(u) \wedge (c,b) \notin T(x)} \{d_{x,c} + w_{c,b}\} & if\ a = x \\ w_{a,b} & otherwise \end{cases}$$

  where $T_x(u)$ is the subtree of $T(x)$ rooted at u.

It is not difficult to see that a shortest paths tree of G_u rooted at x yields information about the shortest distances from x to any descendent y of u in $T(x)$ obtained by excluding from G the edges on the portion of $\pi_{x,y}$ that falls in $B_x(i, j)$.

Claim 1 *If Dijkstra's algorithm is used as subroutine, then algorithm* `exclude-d` *requires* $O(m + n \log n)$ *time in the worst case.*

Proof. The proof easily follows by observing that the sums of the cardinalities of each V_u and E_u are linear in n and m, respectively. □

3 Oracles with query time $O(\log n)$ and space $O(n^2 \log n)$

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

Data structure. Using $O(n^2 \log n)$ space, we maintain each $d_{x,y}$ and $h_{x,y}$, and we maintain four matrices dl, dr, vl, and vr of size $n \times n \times \log n$ defined for any x, y and for any i, $0 \le i < \log n$, as follows:

- $dl[x, y, i]$ = shortest distance from vertex x to vertex y in G obtained by temporary excluding the edges on the sub-path of $\pi_{x,y}$ between level $\lfloor 2^{i-1} \rfloor$ and level 2^i in $T(x)$;
- $dr[x, y, i]$ = shortest distance from vertex y to vertex x in $\widehat{G}$ obtained by temporary excluding the edges on the sub-path of $\widehat{\pi}_{y,x}$ between level $\lfloor 2^{i-1} \rfloor$ and level 2^i in $\widehat{T}(y)$;
- $vl[x, y, i]$ = vertex of $\pi_{x,y}$ that is at level $\lfloor 2^{i-1} \rfloor$ in $T(x)$;
- $vr[x, y, i]$ = vertex of $\widehat{\pi}_{y,x}$ that is at level $\lfloor 2^{i-1} \rfloor$ in $\widehat{T}(y)$.

Preprocessing. Distances $d_{x,y}$ and $h_{x,y}$ and matrices vl and vr are easily initialized from shortest paths trees of G. To compute $dl[x, y, i]$, we call procedure `exclude`$(G, T(x), \lfloor 2^{i-1} \rfloor$, $2^i)$, discussed in Section 2, for each x and for each $i, 0 \le i < \log n$. To compute $dr[x, y, i]$, we call procedure `exclude`$(\widehat{G}, \widehat{T}(y), \lfloor 2^{i-1} \rfloor, 2^i)$, for each x and for each $i, 0 \le i < \log n$.

Query. The main idea of the algorithm is to decrease progressively an upper bound d on the requested distance by using information in dl and dr. The query procedure is as follows:

---

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

1. **if** $d_{x,u} + w_{u,v} + d_{v,y} > d_{x,y}$ **then return** $d_{x,y}$
2. $d \leftarrow +\infty$
3. **if** $h_{x,v} \leq \lceil h_{x,y}/2 \rceil$ **then**
4. $l \leftarrow x$
5. $r \leftarrow vl[x, y, \lceil \log_2 h_{x,v} \rceil + 1]$
6.   **repeat**
7. $i \leftarrow \lceil \log_2 h_{l,v} \rceil$
8. $j \leftarrow \lceil \log_2 h_{u,r} \rceil$
9. $d \leftarrow \min\{d,\ d_{x,l} + dl[l, y, i],\ dr[x, r, j] + d_{r,y}\}$
10. $l \leftarrow vl[l, y, i]$
11. $r \leftarrow vr[x, r, j]$
12.   **until** $(l = u) \wedge (r = v)$
13. **else** { similar }
14. **return** d

Line 3 checks if (u, v) falls in the first or in the second half of $\pi_{x,y}$, and then similar actions are done. Lines 4–5 let $l = x$ and pick $r \in \pi_{x,y}$ such that $h_{x,r}$ is the smallest power of two for which $\pi_{x,r}$ encloses (u, v). Lines 6–12 do a refinement process where l and r jump logarithmically many times up to reach u and v, respectively. During this process, d is progressively decreased from $+\infty$ to the correct answer.

Analysis. To prove the correctness of `distance`, we show that the answer takes into account all possible configurations of the end-points of detours $\pi_{a,b}^{u,v}$ (see Figure 1). To do this, we use a geometric argument: we model paths $\pi_{x,y}$ as segments $[x, y]$ of width $h_{x,y}$ and we say that, for any $x \leq l_1 \leq l_2 \leq r_1 \leq r_2 \leq y$, d *covers* $[l_1, l_2] \times [r_1, r_2]$ if and only if $d \leq$ shortest distance from x to y excluding $[l_2, r_1]$ and using detours $\pi_{a,b}^{u,v}$ with end-points a and b in $[l_1, l_2]$ and $[r_1, r_2]$, respectively. It holds that, if d_1 covers $[l_1, l_2] \times [r_1, r_2]$ and d_2 covers $[l_3, l_4] \times [r_3, r_4]$, then $\min\{d_1, d_2\}$ covers $([l_1, l_2] \times [r_1, r_2]) \cup ([l_3, l_4] \times [r_3, r_4])$. Our goal is to prove that the value d returned by `distance` covers $[x, u] \times [v, y]$. To this aim, we decompose $[x, u] \times [v, y]$ as union of $k \leq \log n$ products of segments and we show that variable d of the algorithm progressively covers all products in this decomposition.

Claim 2 *Let $[l_i, l'_i]$ and $[r_i, r'_i]$ be any two sequences of $k+1$ segments such that $[x, y] = [l_0, l'_0] \supseteq [l_1, l'_1] \supseteq \cdots \supseteq [l_k, l'_k] = [u, v]$ and $[x, y] = [r_0, r'_0] \supseteq [r_1, r'_1] \supseteq \cdots \supseteq [r_k, r'_k] = [u, v]$, for any $x \leq u \leq v \leq y$. Moreover, let $L_i = [l_i, l_{i+1}] \times [l'_{i+1}, y]$, $R_i = [x, r_{i+1}] \times [r'_{i+1}, r'_i]$, $S_0 = \emptyset$, and $S_{i+1} = L_i \cup R_i \cup S_i$. If $l_i \leq r_i$ and $l'_i \leq r'_i$ for each $0 \leq i \leq k$, then $S_i \supseteq [x, l_i] \times [r'_i, y]$.*

Proof. The proof follows by induction (see Figure 2):

$$S_{i+1} \supseteq ([l_i, l_{i+1}] \times [l'_{i+1}, y]) \cup ([x, r_{i+1}] \times [r'_{i+1}, r'_i]) \cup ([x, l_i] \times [r'_i, y]) \supseteq$$

$$\supseteq ([l_i, l_{i+1}] \times [r'_{i+1}, y]) \cup ([x, l_{i+1}] \times [r'_{i+1}, r'_i]) \cup ([x, l_i] \times [r'_i, y]) = [x, l_{i+1}] \times [r'_{i+1}, y]. \quad \square$$

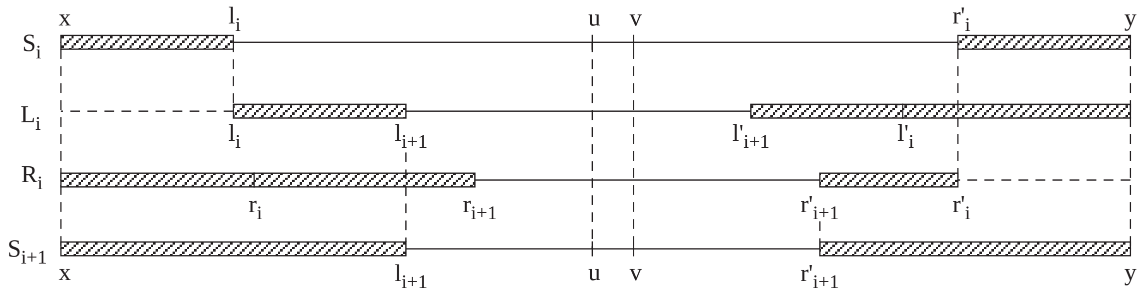


Figure 2: Proof of Claim 2.

Claim 2 implies that $S_k = \bigcup_{i=0}^{k-1}(L_i \cup R_i) = [x,u] \times [v,y]$, i.e., S_i's form a decomposition of $[x,u] \times [v,y]$. Now, let q_i and q'_i be two sequences defined as:

$$q_0 = x, \qquad q_{i+1} = \begin{cases} q_i + 2^{\lceil \log(v-q_i) \rceil - 1} & \text{if } v - q_i > q'_i - u \\ q_i & \text{otherwise} \end{cases}$$

$$q'_0 = 2^{\lceil \log v \rceil}, \qquad q'_{i+1} = \begin{cases} q'_i - 2^{\lceil \log(q'_i-u) \rceil - 1} & \text{if } v - q_i < q'_i - u \\ q'_i & \text{otherwise} \end{cases}$$

It is not difficult to see that, if we remove duplicates from q_i and from q'_i, these sequences correspond to the successive values taken in `distance` by variables l and r, respectively. We also observe that d progressively covers in line 9 products $[q_i, q_{i+1}] \times [q_i + 2^{\lceil \log(v-q_i) \rceil}, y]$ and $[x, q'_i - 2^{\lceil \log(q'_i-u) \rceil}] \times [q'_{i+1}, q'_i]$, accessing matrices dl and dr, respectively. Moreover, we can prove that, if we let $[l_i, l'_i] = [q_i, q_i + 2^{\lceil \log(v-q_i) \rceil}]$ and $[r_i, r'_i] = [q'_i - 2^{\lceil \log(q'_i-u) \rceil}, q'_i]$, then $[l_i, l'_i]$ and $[r_i, r'_i]$ satisfy the hypotheses of Claim 2 (we defer the details to the full paper). Thus, eventually d covers $[x,u] \times [v,y]$.

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

4 Oracles with query time $O(1)$ and space $O(n^{2.5})$

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 query time and preprocessing time down to $O(1)$ and down to $O(mn^{1.5} + n^{2.5} \log n)$, respectively. This second oracle uses $O(n^{2.5})$ space.

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

- $dh[x,y,i]$ = shortest distance from vertex x to vertex y in G obtained by temporary excluding the i-th edge among the $\sqrt{n}$ edges at the head of $\pi_{x,y}$;
- $dt[x,y,i]$ = shortest distance from vertex x to vertex y in G obtained by temporary excluding the i-th edge among the $\sqrt{n}$ edges at the tail of $\pi_{x,y}$;
- $dc[x,y,i]$ = shortest distance from vertex x to vertex y in G obtained by temporary excluding the edges on the sub-path between the q_i-th vertex and the q_{i+1}-th vertex of $\pi_{x,y}$, where $q_0 = 0 < q_1 < \cdots < q_k < n$ is any increasing sequence of $k = O(\sqrt{n})$ positive numbers depending on x, and $q_i - q_{i-1} \le \sqrt{n}$ for any i, $1 < i \le k$;

- $vc[x, y, i]$ = vertex of $\pi_{x,y}$ at level q_i in $T(x)$;
- $bc[x, y]$ = index i such as y is at a level between q_i and q_{i+1} in $T(x)$.

Preprocessing. As distances $d_{x,y}$ and $h_{x,y}$ are easily initialized from shortest paths trees of G, we focus on preprocessing matrices dh, dt, dc, vc, and bc. Since bands $B_x(i, i+1)$ contain paths formed by single edges, preprocessing matrix dh can be done by performing calls to algorithm `exclude-d`(G, $T(x)$, i, $i+1$), presented in Section 2, for each vertex x and for each i such that $1 \leq i < \sqrt{n}$. Similarly, matrix dt can be initialized via calls to algorithm `exclude-d`($\widehat{G}$, $\widehat{T}(y)$, i, $i+1$) for each vertex y and for each i such that $1 \leq i < \sqrt{n}$. We remark that, since we deal with bands of edge-disjoint paths, we can use `exclude-d` instead of `exclude`.

To compute dc, we consider the problem of cutting each shortest paths tree $T(x)$ into $O(\sqrt{n})$ bands of edge-disjoint paths, such that each band has height $\leq \sqrt{n}$. More in detail, for any vertex x we wish to find a sequence $q_0 = 0 < q_1 < \cdots < q_k < n$ of length $k = O(\sqrt{n})$ such that $B_x(q_i, q_{i+1})$ contains only edge-disjoint paths in $T(x)$ and $q_i - q_{i-1} \leq \sqrt{n}$ for any i, $1 < i \leq k$. The following claim provides a nice combinatorial property on trees that helps us to solve the problem.

Claim 4 *Let T be a tree with n vertices and let $size(v)$ be the number of vertices in the subtree of T rooted at v. If T' is obtained from T by deleting any vertex u such that $size(u) \leq \sqrt{n}$, then at most $\sqrt{n}$ vertices in T' have out-degree > 1.*

Proof. Since T' contains only vertices that have size $> \sqrt{n}$ 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 3a). By Claim 4, $k \leq \sqrt{n}$. An interesting fact is that cutting T at each l_i yields bands of edge-disjoint paths:

Claim 5 *For any $1 \leq i < k$, $B(l_i, l_{i+1})$ is a band of edge-disjoint paths in T.*

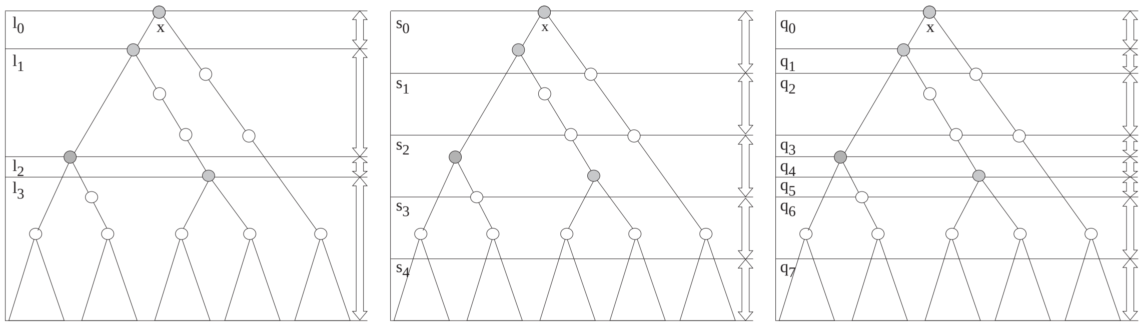
Proof. We recall that $B(l_i, l_{i+1})$ contains all the paths that connect vertices at level l_i with vertices at level l_{i+1} in T. The proof easily follows by observing that, by the definition of sequence $\{l_i\}$, all vertices in T at levels $l_i + 1$ to $l_{i+1} - 1$ have out-degree ≤ 1. □

Unfortunately, we cannot use directly sequences $\{l_i\}$, defined for each $T(x)$, as sequences $\{q_i\}$ for computing dc, since we are not guaranteed that $l_i - l_{i-1} \leq \sqrt{n}$. To solve this problem, we note that splitting a band of edge-disjoint paths yields again bands of edge-disjoint paths:

Claim 6 *If $B(i, j)$ is a band of edge-disjoint paths in T, then for any $i < h < j$, both $B(i, h)$ and $B(h, i)$ are bands of edge-disjoint paths in T.*

Let $s_0 < s_1 < \cdots < s_{\sqrt{n}}$ be a sequence such that $s_i = i \cdot \sqrt{n}$ (Figure 3b). By Claim 6, if we merge sequences $\{l_i\}$ and $\{s_i\}$, we obtain an ordered sequence $\{q_i\}$ of length $\leq 2\sqrt{n}$ with the desired properties (Figure 3c). We remark that $\{q_i\}$ induces at most $2\sqrt{n}$ bands of edge-disjoint paths with height $\leq \sqrt{n}$. To initialize dc, we can thus perform calls to `exclude-d`$(G, T(x), q_i, q_{i+1})$ for each vertex x and for each $1 \leq i < 2\sqrt{n}$. Again, we can use `exclude-d` instead of `exclude`.

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



(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 3: Preprocessing matrix dc: cutting trees to form bands of edge-disjoint paths

Query. The query procedure is as follows:

---

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

1. **if** $d_{x,u} + w_{u,v} + d_{v,y} > d_{x,y}$ **then return** $d_{x,y}$
2. $i \leftarrow bc[x, u]$
3. $\hat{u} \leftarrow vc[x, y, i]$
4. $\hat{v} \leftarrow vc[x, y, i+1]$
5. $d \leftarrow \min\{\ dc[x, y, i],\ d_{x,\hat{u}} + dh[\hat{u}, y, h_{\hat{u},u}],\ dt[x, \hat{v}, h_{v,\hat{v}}] + d_{\hat{v},y}\ \}$
6. **return** d

In line 2 we retrieve the unique index i such that (u, v) falls in $B_x(q_i, q_{i+1})$, and then in lines 3–4 we identify the vertices $\widehat{u}$ and $\widehat{v}$ on the path $\pi_{x,y}$ in $T(x)$ that are at levels q_i and q_{i+1}, respectively. The correct answer is given in line 5 by accessing matrices dc, dh, and dt.

Analysis. To prove the correctness of `distance`, we show that the answer takes into account all possible configurations of the end-points of detours $\pi^{u,v}_{a,b}$ (see Figure 1). As in the analysis of the oracle presented in Section 3, we use a geometric argument: we model paths $\pi_{x,y}$ as segments $[x, y]$ of width $h_{x,y}$ and we say that, for any $x \le l_1 \le l_2 \le r_1 \le r_2 \le y$, d *covers* $[l_1, l_2] \times [r_1, r_2]$ if and only if $d \le$ shortest distance from x to y excluding $[l_2, r_1]$ and using detours $\pi^{u,v}_{a,b}$ with end-points a and b in $[l_1, l_2]$ and $[r_1, r_2]$, respectively. Our goal is to prove that the value d returned by `distance` covers $[x, u] \times [v, y]$. To this aim, we decompose $[x, u] \times [v, y]$ as union of 3 products of segments and we show that variable d of the algorithm covers the products in this decomposition.

Claim 7 *For any $x \le \widehat{u} \le u \le v \le \widehat{v} \le y$, it holds that:*

$$[x, u] \times [v, y] \ = \ [x, \widehat{u}] \times [\widehat{v}, y] \ \cup \ [\widehat{u}, u] \times [v, y] \ \cup \ [x, u] \times [v, \widehat{v}]$$

It is easy to see that:

- $dc[x, y, i]$ covers $[x, \widehat{u}] \times [\widehat{v}, y]$: this is the correct answer for `distance`(x, y, u, v) in the case where both the end-points of the detour $\pi^{u,v}_{a,b}$ fall outside the band $B_x(q_i, q_{i+1})$ (see Figure 4b).

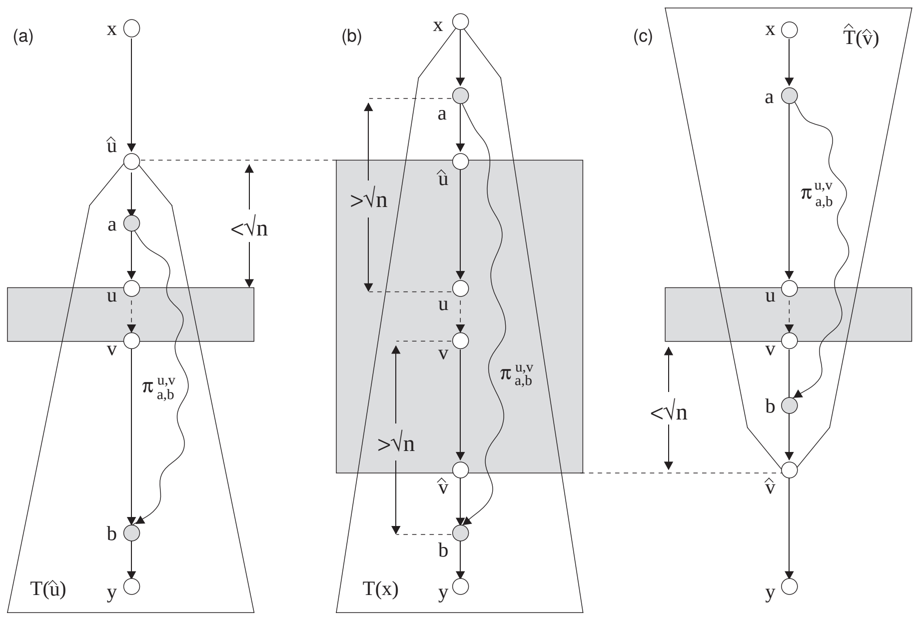


Figure 4: Possible positions of the end-points of the detour $\pi^{u,v}_{a,b}$ with respect to the excluded edge (u, v).

- $d_{x,\hat{u}}+dh[\hat{u}, y, h_{\hat{u},u}]$ covers $[\widehat{u}, u]\times[v, y]$: this is the correct answer for `distance`(x, y, u, v) in the case where the end-point a of the detour $\pi^{u,v}_{a,b}$ falls inside the band $B_x(q_i, q_{i+1})$ (see Figure 4a).
- $dt[x, \hat{v}, h_{v,\hat{v}}]+d_{\hat{v},y}$ covers $[x, u]\times[v, \widehat{v}]$: this is the correct answer for `distance`(x, y, u, v) in the case where the end-point b of the detour $\pi^{u,v}_{a,b}$ falls inside the band $B_x(q_i, q_{i+1})$ (see Figure 4c).

Thus, by Claim 7, d covers $[x, u] \times [v, y]$.

Claim 8 *Preprocessing requires $O(mn^{1.5} + n^{2.5} \log n)$ worst-case time and any* `distance` *operation requires $O(1)$ worst-case time.*

Proof. Growing shortest paths trees $T(x)$ for all vertices x requires $O(mn + n^2 \log n)$ in the worst-case [3]. The proof for the preprocessing follows from Claim 1 by observing that initializing dc, dh, and dt is carried out via $O(\sqrt{n})$ calls to `exclude-d` for each vertex x. The bound for queries is straightforward. □

References

[1] 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.

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

[3] 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.

[4] 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)*, pages 81–99, 1999.

[5] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proc. 31st ACM Symposium on Theory of Computing (STOC'99)*, pages 492–498, 1999.

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

[7] U. Zwick. All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms. In *Proc. of the 39th IEEE Annual Symposium on Foundations of Computer Science (FOCS'98)*, pages 310–319, Los Alamitos, CA, November 8–11 1998.