# Improved Distance Oracles for Avoiding Link-Failure*

Rezaul Alam Chowdhury and Vijaya Ramachandran
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
shaikat@cs.utexas.edu, vlr@cs.utexas.edu

TR-02-35

June 28, 2002

**Abstract**

We consider the problem of preprocessing an edge-weighted directed graph to answer queries that ask for the shortest path from any given vertex to another avoiding a failed link. We present two algorithms that improve on earlier results for this problem. Our first algorithm, which is a modification of an earlier method, improves the query time to a constant while maintaining the earlier bounds for preprocessing time and space. Our second result is a new algorithm whose preprocessing time is considerably faster than earlier results and whose query time and space are worse by no more than a logarithmic factor.

## 1 Introduction

Given an edge-weighted directed graph $G = (V, E, w)$, where $w$ is a weight function on $E$, the *distance sensitivity problem* asks for the construction of a data structure called the *distance sensitivity oracle* that supports any sequence of the following two queries:

- distance$(x, y, u, v)$: return the shortest distance from vertex $x$ to vertex $y$ in $G$ avoiding the edge $(u, v)$.
- path$(x, y, u, v)$: return the shortest path from vertex $x$ to vertex $y$ in $G$ avoiding the edge $(u, v)$.

This problem as formulated above was first addressed by Demetrescu and Thorup in [DT02] for directed graphs with nonnegative real valued edge weights. In an earlier paper, King and Sagert [KS99] addressed a variant of this problem related to reachability in directed acyclic graphs.

As in [DT02], in this paper we concentrate on answering distance queries in digraphs with nonnegative real valued edge weights under the failure of a single link. The goal is to preprocess the graph in order to answer distance queries quickly when a link failure is detected. It is assumed that the time gap between two successive link failures is long enough to permit us to compute a new data structure in the background that will assist us in answering distance queries when the next link failure is detected.

*This work was supported in part by Texas Advanced Research Program Grant 003658-0029-1999 and NSF Grant CCR-9988160. Chowdhury was also supported by an MCD Graduate Fellowship.

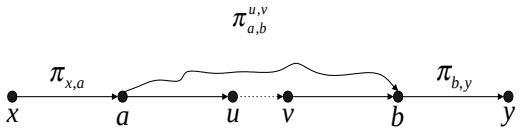


Figure 1: Optimal detour [DT02].

In this paper we present two new algorithms for the distance sensitivity problem. An overview of our results is given in Section 4. In Section 2 we describe the notation we use, and we summarize earlier results in Section 3. In Sections 5 and 6 we present details of our two new methods.

## 2 Notations

Most of the notation we use is from [DT02], which we include below for convenience. By $n$ and $m$ we denote respectively the number of vertices and the number of edges in $G$, i.e., $n = |V|$ and $m = |E|$. By $T(x)$ we denote the shortest-path tree rooted at vertex $x$ of $G$ and by $\pi_{x,y}$ we denote the unique $x$ to $y$ path in $T(x)$. As in [DT02] we assume that all shortest paths in $G$ are unique, since if not, we can make them unique by adding small random fractions to the edge weights. By $h_{x,y}$ we denote the number of edges in $\pi_{x,y}$. The weight of any edge $(x, y)$ will be represented by $w_{x,y}$ and the weighted shortest distance from $x$ to $y$ will be represented by $d_{x,y}$. We denote by $\hat{G}$ the graph obtained by reversing the orientation of the edges in $G$ and $\hat{\pi}$ and $\hat{T}$ respectively represent $\pi$ and $T$ related to $\hat{G}$. Since we assume that all shortest paths are unique, $\pi_{x,y}$ and $\hat{\pi}_{y,x}$ will have the same set of edges and for any vertex $z$ on the shortest path from $x$ to $y$, $\pi_{x,z}, \pi_{z,y} \subseteq \pi_{x,y}$.

Let $u, v$ be vertices on a shortest path from $x$ to $y$, with $u$ being closer to $x$ than $v$. By $\pi^{u,v}_{x,y}$ we denote a shortest path from vertex $x$ to vertex $y$ in $G$ that avoids the sub-path from $u$ to $v$ (this is a slight generalization of the notation used in [DT02]). Let $a$ be a vertex on $\pi_{x,u}$ and $b$ a vertex on $\pi_{v,y}$ such that $\pi^{u,v}_{x,y} = \pi_{x,a} \cdot \pi^{u,v}_{a,b} \cdot \pi_{b,y}$ and $\pi_{a,b} \cap \pi^{u,v}_{a,b} = \emptyset$, where $\cdot$ denotes the path concatenation operator. So $\pi^{u,v}_{a,b}$ is the *best detour* one can follow in order to go from $x$ to $y$ while avoiding the subpath from $u$ to $v$, and $\pi_{x,a}$ and $\pi_{b,y}$ represent the longest common portions of $\pi_{x,y}$ and $\pi^{u,v}_{x,y}$ (see Figure 1). Now consider vertices $l_1, l_2, r_1, r_2$ that lie on $\pi_{x,y}$ at increasing distances from $x$. Let $d$ be an upper bound on the length of $\pi^{l_2,r_1}_{x,y}$. The value $d$ is said to *cover* $[l_1, l_2] \times [r_1, r_2]$ if $d \leq$ shortest distance from $x$ to $y$ avoiding the interval $[l_2, r_1]$ when using a detour with $a$ in $\pi_{l_1,l_2}$ and $b$ in $\pi_{r_1,r_2}$.

Let $P$ be a set of shortest paths in $G$. The paths in $P$ are *independent* in $G$ w.r.t. $x$ if for any $\pi_1, \pi_2 \in P$ no edge of $\pi_1$ is a descendent of $\pi_2$ in $T(x)$ and vice versa. In our algorithms we will make use of two procedures *exclude* and *exclude-d* from [DT02]. If $P$ is a set of shortest paths in $G$, both *exclude*$(G, x, P)$ and *exclude-d*$(G, x, P)$ compute, for each path $\pi \in P$ the shortest distance from vertex $x$ to all other vertices in $G$ avoiding the edges on $\pi$. The procedure *exclude*$(G, x, P)$ runs in $O(|P|(m + n \log n))$ worst-case time using a fast Dijkstra computation [FT87] for each $\pi \in P$. On the other hand, if the shortest paths in $P$ are independent, *exclude-d* can perform the same computation in $O(m + n \log n)$ worst-case time.

## 3 Existing Algorithms

A straight-forward approach to solving the distance sensitivity problem is to use a recent dynamic all pairs shortest paths (APSP) algorithm [DI01, K99, KT01] and delete a *specific* failed link. The time

required is quite high ($\tilde{O}(n^{2.5})$ [1] amortized) even for unweighted graphs though after that the queries can be answered in $O(1)$ time assuming failure of that specific link. In contrast, the distance sensitivity problem asks for a preprocessing of the graph so that queries can be answered quickly under failure of *any* one link.

Another straightforward approach is to construct a table of size $O(n^3)$ that will store for each vertex pair $(x, y)$ the shortest distance from $x$ to $y$ avoiding each of the $O(n)$ edges on the shortest $x$ to $y$ path in the original graph. The total time required to construct the table is $\tilde{O}(mn^2)$ [T01] and the query time is $O(1)$. The problem with this approach is the excessive space requirement (and its relatively large preprocessing time).

The first two non-trivial algorithms for solving the distance sensitivity problem were presented by Demetrescu and Thorup [DT02]. For convenience we name these two algorithms as DT-1 and DT-2, and summarize their complexities in Table 1. In DT-1 each $x$ to $y$ shortest path is divided into $O(\log n)$ segments and the shortest distance from $x$ to $y$ avoiding each of these $O(\log n)$ segments is stored in a table. The procedure *exclude* is used to compute these distances. The query algorithm progressively decreases an upper bound on the shortest distance from $x$ to $y$ avoiding a failed link $(u, v)$ and returns the correct answer in $O(\log n)$ iterations. In DT-2, each shortest-path tree $T(x)$ is divided into $O(\sqrt{n})$ bands of independent paths. For each of the bands in each $T(x)$ the procedure *exclude-d* is used to compute the shortest distances from $x$ to all $y$'s avoiding the paths in that band. After this preprocessing, queries can be answered in constant time.

## 4 Our Results

In this paper, we present two algorithms: CR-1 and CR-2. The algorithm CR-1 is a simple but useful modification of DT-1 which brings down the query time to constant while leaving the other costs unchanged. The second algorithm (CR-2) is based on a new approach for finding bands of independent paths that reduces the preprocessing time significantly without increasing the other costs by more than a logarithmic factor. More precisely, when compared with DT-1, CR-2 reduces the preprocessing time by a factor of $\frac{n}{\log n}$ at the cost of increasing the space requirement by a factor of $\log n$. The query time is unchanged. When compared with DT-2, CR-2 reduces the preprocessing time by a factor of $\frac{\sqrt{n}}{\log n}$ at the cost of increasing the query time by a factor of $\log n$. In this case the space requirement is also reduced significantly — by a factor of $\frac{\sqrt{n}}{\log^2 n}$.

Table 1 : Distance Sensitivity Oracles in [DT02]

| Algorithm | Preprocessing Time | Extra Space | Query Time |
|---|---|---|---|
| DT-1 | $O(mn^2 \log n + n^3 \log^2 n)$ | $O(n^2 \log n)$ | $O(\log n)$ |
| DT-2 | $O(mn^{1.5} \log n + n^{2.5} \log^2 n)$ | $O(n^{2.5})$ | $O(1)$ |

Table 2 : Our Contribution

| Algorithm | Preprocessing Time | Extra Space | Query Time |
|---|---|---|---|
| CR-1 | $O(mn^2 \log n + n^3 \log^2 n)$ | $O(n^2 \log n)$ | $O(1)$ |
| CR-2 | $O(mn \log^2 n + n^2 \log^3 n)$ | $O(n^2 \log^2 n)$ | $O(\log n)$ |

## 5 The Algorithm CR-1

CR-1 is a simple modification of DT-1 which improves the query time to $O(1)$ while keeping the preprocessing time and the space requirement unchanged ($O(mn^2 \log n + n^3 \log^2 n)$ and $O(n^2 \log n)$ respectively). The only difference in preprocessing between CR-1 and DT-1 is the use of two additional

[1] $\tilde{O}(f(n))$ is used to denote $O(f(n)\text{polylog}(n))$.

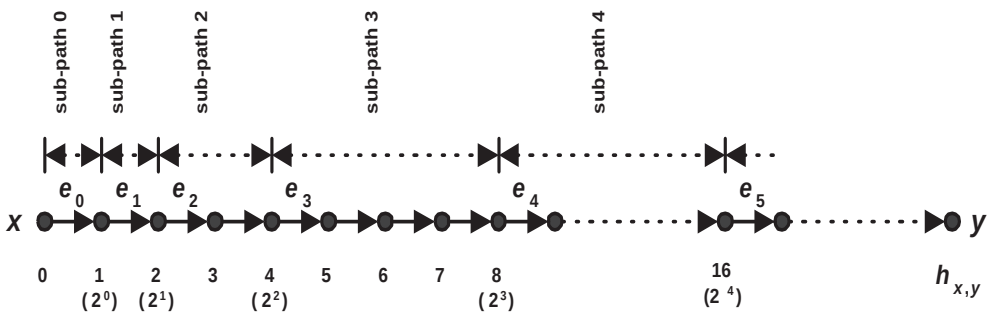


Figure 2: Edge $e_i$ is the edge on $\pi_{x,y}$ that is excluded in calculating the distance $sl(x, y, i)$. Sub-path $i$ is the sub-path excluded in calculating $dl(x, y, i)$.

matrices, $sl$ and $sr$, in CR-1. The query algorithm in CR-1 uses these two matrices to reduce the query time to constant (from $\Theta(\log n)$ in DT-1). The matrix $sl$ is similar to the matrix $dl$ in DT-1: while $dl(x, y, i)$ stores the shortest distance from vertex $x$ to $y$ in $G$ without the edges of the sub-path of $\pi_{x,y}$ between level $\lfloor 2^{i-1} \rfloor$ and level $2^i$ in $T(x)$, $sl(x, y, i)$ stores the $x$ to $y$ shortest distance avoiding only the first edge (i.e., edge nearest to $x$) on that sub-path. A similar relation holds between matrices $dr$ and $sr$. Combined with the data stored in $dl$ and $dr$, the $sl$ and $sr$ matrices allow us to answer the query $distance(x, y, u, v)$ in constant time.

**Data Structure**

We maintain each $d_{x,y}$ and $h_{x,y}$ using $O(n^2)$ space. We use $O(n^2 \log n)$ space in order to maintain six matrices $dl$, $dr$, $sl$, $sr$, $vl$ and $vr$ of size $n \times n \times \log n$ each defined for any $x$, $y$ and for any $i$, $0 \leq i < \log n$, as follows:

- $dl[x, y, i]$ = shortest distance from vertex $x$ to vertex $y$ in $G$ without 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 $\hat{G}$ without the edges on the sub-path of $\hat{\pi}_{y,x}$ between level $\lfloor 2^{i-1} \rfloor$ and level $2^i$ in $\hat{T}(y)$;
- $sl[x, y, i]$ = shortest distance from vertex $x$ to vertex $y$ in $G$ without the edge of $\pi_{x,y}$ between level $\lfloor 2^{i-1} \rfloor$ and level $\lfloor 2^{i-1} \rfloor + 1$ in $T(x)$;
- $sr[x, y, i]$ = shortest distance from vertex $y$ to vertex $x$ in $\hat{G}$ without the edge of $\hat{\pi}_{y,x}$ between level $\lfloor 2^{i-1} \rfloor$ and level $\lfloor 2^{i-1} \rfloor + 1$ in $\hat{T}(y)$;
- $vl[x, y, i]$ = vertex of $\pi_{x,y}$ at level $\lfloor 2^{i-1} \rfloor$ in $T(x)$;
- $vr[x, y, i]$ = vertex of $\hat{\pi}_{y,x}$ at level $\lfloor 2^{i-1} \rfloor$ in $\hat{T}(y)$;

**Preprocessing**

As in [DT02], we initialize the distances $d_{x,y}$ and $h_{x,y}$ and the matrices $vl$ and $vr$ from the shortest-path trees of $G$, and the matrices $dl$ and $dr$ by calling the procedure *exclude*.

Let $B_{T(x)}(j,k)$ denote the band of paths in $T(x)$ that connect vertices at level $j$ with vertices at level $k > j$ in the tree assuming that $x$ is at level 0 in $T(x)$. For each $x$ and for each $i$, $0 \leq i < \log n$, we compute $sl[x,y,i]$ by calling procedure *exclude-d*$(G, x, B_{T(x)}(\lfloor 2^{i-1} \rfloor, \lfloor 2^{i-1} \rfloor + 1))$. We compute $sr[x,y,i]$ by calling procedure *exclude-d*$(\hat{G}, y, B_{\hat{T}(y)}(\lfloor 2^{i-1} \rfloor, \lfloor 2^{i-1} \rfloor + 1))$, for each $y$ and for each $i$, $0 \leq i < \log n$.

**Query**

The query algorithm is as follows:

```
 function distance(x, y, u, v):  ℜ
1.   if d_{x,u} + w_{u,v} + d_{v,y} > d_{x,y} then return d_{x,y} fi
2.   if x = u then return sl[x,y,0] fi
3.   if y = v then return sr[x,y,0] fi
4.   l := ⌈log_2 h_{x,u}⌉
5.   if l = log_2 h_{x,u} then return sl[x,y,l+1] fi
6.   r := ⌈log_2 h_{v,y}⌉
7.   if r = log_2 h_{v,y} then return sr[x,y,r+1] fi
8.   p := vr[x,u,l], q := vl[v,y,r]
9.   d := min(d_{x,p} + sl[p,y,l], sr[x,q,r] + d_{q,y})
10.  if h_{x,u} ≤ h_{v,y} then
11.     d := min(d, dl[x,y,l])
12.   else
13.     d := min(d, dr[x,y,r])
14.  fi
15.  return d
```

**Correctness**

In line 1 of the query algorithm, we get rid of the case where $(u,v) \notin \pi_{x,y}$ and return $d_{x,y}$ as the answer. Lines 2 and 3 handle the cases where $(u,v)$ is the first or the last edge on $\pi_{x,y}$. Lines 4 and 5 take care of the case where $(u,v)$ is at a distance $2^l$ from vertex $x$ on $\pi_{x,y}$ for some nonnegative integer $l$, $0 \leq l < \log n$. Lines 6 and 7 handle the case where $(u,v)$ is at a distance $2^r$ from vertex $y$ on $\hat{\pi}_{y,x}$ for some nonnegative integer $r$, $0 \leq l < \log n$. Lines 8 to 15 take care of the remaining cases.

Lines 2 to 7 handle the following four trivial cases: (1) $h_{x,u} = 0$, (2) $h_{v,y} = 0$, (3) $h_{x,u} = 2^l$ for some nonnegative integer $l$, $0 \leq l < \log n$, and (4) $h_{v,y} = 2^r$ for some nonnegative integer $r$, $0 \leq r < \log n$. So in order to prove the correctness of *distance* we need only to prove the correctness of the code segment of lines 8 to 14 that handles the nontrivial case when none of the above four conditions hold.

Let $d_1 = d_{x,p} + sl[p,y,l]$. The distance $d_1$ covers $[p,u] \times [v,y]$ and the distance $d_2 = sr[x,q,r] + d_{q,y}$ covers $[x,u] \times [v,q]$. Now let us consider the case when $h_{x,u} \leq h_{v,y}$. In this case $0 \neq h_{x,p} < h_{p,u} = 2^{l-1} \leq h_{v,q}$. $dl[x,y,l]$ is the distance from $x$ to $y$ avoiding a sub-path of length $2^{l-1}$ at a distance of $2^{l-1}$ from $x$ on $\pi_{x,y}$ . Let the endpoints of that sub-path be $p'$ and $q'$, and $h_{x,p'} < h_{x,q'}$. So $[p',q']$ contains $(u,v)$ and $[p,q]$ contains $[p',q']$. Hence, $dl[x,y,l]$ covers $[x,p] \times [q,y]$. But $[p,u] \times [v,y] \cup [x,u] \times [v,q] \cup [x,p] \times [q,y] = [x,u] \times [v,y]$. So $d$ covers $[x,u] \times [v,y]$. Similar argument holds for the case when $h_{x,u} > h_{v,y}$. Hence we conclude

CLAIM 5.1. *The query function distance(x, y, u, v) for algorithm CR-1 correctly computes the shortest distance from vertex $x$ to vertex $y$ in $G$ avoiding the edge $(u,v)$.*

### Preprocessing, Query and Space Bounds

CLAIM 5.2. *In algorithm CR-1, the preprocessing requires $O(mn^2 \log n + n^3 \log^2 n)$ worst-case time, the data structure requires $O(n^2 \log n)$ space, and any distance query can be answered in $O(1)$ worst-case time.*

*Proof.* The entries of the matrices $dl$ and $dr$ are computed as in [DT02] in $O(mn^2 \log n + n^3 \log^2 n)$ worst-case time. For each $x$ and for each $i$, $0 \leq i < \log n$, we can compute $sl[x, y, i]$ by calling procedure *exclude-d*$(G, x, B_{T(x)}(\lfloor 2^{i-1} \rfloor, \lfloor 2^{i-1} \rfloor + 1))$ since the single-edge paths in $B_{T(x)}(\lfloor 2^{i-1} \rfloor, \lfloor 2^{i-1} \rfloor + 1)$ are trivially independent. Since *exclude-d*$(G, x, B_{T(x)}(\lfloor 2^{i-1} \rfloor, \lfloor 2^{i-1} \rfloor + 1))$ 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 $O(mn \log n + n^2 \log^2 n)$. Hence the preprocessing time is dominated by the time to compute the $dl$ and $dr$ matrices and requires $O(mn^2 \log n + n^3 \log^2 n)$ worst-case time.

The query algorithm does not contain any loops and it does not make any function call either. Therefore it runs in $O(1)$ worst-case time.

The $d$ and $h$ matrices have size $n \times n$, and each of the matrices $dl$, $dr$, $sl$, $sr$, $vl$ and $vr$ has size $n \times n \times \log n$. Therefore the total space requirement is $O(n^2 \log n)$. □

## 6 The Algorithm CR-2

This is a new algorithm with preprocessing time $O(mn \log^2 n + n^2 \log^3 n)$, query time $O(\log n)$ and space $O(n^2 \log^2 n)$. In this algorithm we divide each shortest-path tree $T(x)$ into $O(\log n)$ bands of independent paths. In order to achieve a logarithmic query time we further subdivide each path in a band into $O(\log n)$ segments, thus producing a total of $O(\log^2 n)$ sub-bands (or *segments*) of independent paths in each $T(x)$. For each sub-band of independent paths we use *exclude-d* to compute the required distances to be stored in our data structure. Compared to DT-1, DT-2 and CR-1, this strategy significantly reduces the preprocessing time without increasing the query time or space requirement by more than a factor of $\log n$. (Note that DT-2 divides each $T(x)$ into $O(\sqrt{n})$ bands of independent paths.)

### Data Structure

In the following, each of the matrices $d$, $h$, $bcl$ and $bcr$ has size $O(n^2)$, and each of the matrices $vl$, $vr$, $dl$ and $dr$ has size $O(n^2 \log^2 n)$.

- $bcl[x, y]$ = index $i$ such that the $i^{th}$ segment of $\pi_{x,w}$ contains $y$ where $w$ is a descendent of $y$ in $T(x)$. Defined only for the internal nodes of $T(x)$;
- $bcr[x, y]$ = index $i$ such that the $i^{th}$ segment of $\hat{\pi}_{y,w}$ contains $x$ where $w$ is a descendent of $x$ in $\hat{T}(y)$. Defined only for the internal nodes of $\hat{T}(y)$;
- $vl[x, y, i]$ = first vertex (i.e., vertex nearest from $x$) of $\pi_{x,y}$ contained in the $i^{th}$ segment of $\pi_{x,y}$ in $T(x)$;
- $vr[x, y, i]$ = first vertex (i.e., vertex nearest from $y$) of $\hat{\pi}_{y,x}$ contained in the $i^{th}$ segment of $\hat{\pi}_{y,x}$ in $\hat{T}(y)$;
- $dl[x, y, i]$ = shortest distance from vertex $x$ to vertex $y$ in $G$ without the edges on the $i^{th}$ segment of $\pi_{x,y}$ in $T(x)$;
- $dr[x, y, i]$ = shortest distance from vertex $y$ to vertex $x$ in $\hat{G}$ without the edges on the $i^{th}$ segment of $\hat{\pi}_{y,x}$ in $\hat{T}(y)$;

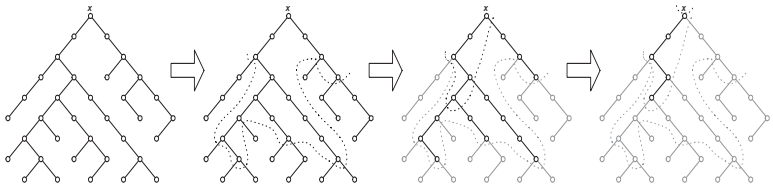

Figure 3: Repeated applications of *Shrink* create bands of independent paths in $T(x)$. In the figure the dark paths below each new dotted curve represent a collection of independent paths.

**Preprocessing**

Distances $d_{x,y}$ and $h_{x,y}$ and matrices $vl$ and $vr$ are easily initialized from shortest-path trees of $G$. However, before describing how the other matrices are initialized we will first describe how the shortest paths are segmented.

A *chain* in a directed graph $G$ is a path $\langle v_1, v_2, \ldots, v_k \rangle$ such that each $v_i$ has exactly one incoming edge and one outgoing edge in $G$. A *maximal chain* is one that cannot be extended. As defined in [R97] a *leaf chain* $\langle v_1, v_2, \ldots, v_{l-1}, v_l \rangle$ in a rooted tree $T$ consists of a maximal chain $\langle v_1, v_2, \ldots, v_{l-1} \rangle$, together with a vertex $v_l$, which is a leaf and the unique child of $v_{l-1}$ in $T$. For convenience we modify this definition slightly by augmenting the leaf chain $\langle v_1, v_2, \ldots, v_l \rangle$ to $\langle v_0, v_1, \ldots, v_l \rangle$ where $v_0$ is the predecessor of $v_1$ in $T$ and call this $\langle v_0, v_1, \ldots, v_l \rangle$ a leaf chain instead of $\langle v_1, v_2, \ldots, v_l \rangle$.

CLAIM 6.1. *The set of all leaf chains of a shortest-path tree $T(x)$ form a band of independent paths with respect to $x$.*

*Proof.* By definition, no two leaf chains share an edge. It also follows from the definition that no edge in one leaf chain is a descendent of another edge in another leaf chain. Therefore the set of all leaf chains of $T(x)$ form a band of independent paths with respect to $x$. □

Let us consider a tree operation *Shrink* [R87] that removes all leaf chains. The following Lemma (with a slightly different wording) has been proved in [R87] (see also [NNS89]).

LEMMA 6.1. *Any $n$ node tree can be transformed into a single vertex with $O(\log n)$ applications of the Shrink operation.*

This leads to the following corollary:

COROLLARY 6.1. *Each shortest-path tree $T(x)$ of $G$ can be partitioned into $O(\log n)$ bands of independent paths with respect to $x$.*

Now consider any leaf chain $\langle v_0, v_1, \ldots, v_l \rangle$ in a band. We divide the chain into $O(\log l)$ segments where the $i^{th}$ $(i \geq 0)$ segment is the sub-chain $\langle v_{s(i)}, v_{s(i)+1}, \ldots, v_{\min(e(i),l)} \rangle$ and this segment exists iff $s(i) < l$. Here $s(i)$ and $e(i)$ are defined as follows:

$$\langle s(i), e(i) \rangle = \begin{cases} \langle i, i+1 \rangle, & if \quad i \leq 1 \\ \langle 2^{\frac{i}{2}}, 3 \times 2^{\frac{i}{2}-1} \rangle, & if \quad i > 1 \quad and \quad even \\ \langle 3 \times 2^{\frac{i-3}{2}}, 2^{\frac{i+1}{2}} \rangle, & if \quad i > 1 \quad and \quad odd \end{cases}$$

Since we know that splitting any band of independent paths yields again bands of independent paths, we can split each of the $O(\log n)$ bands created using the *Shrink* operation, into $O(\log n)$ sub-bands of independent paths where the $i^{th}$ $(i \geq 0)$ sub-band includes the $i^{th}$ segment (if it exists) of each of the sub-paths in that band (see Figure 4).

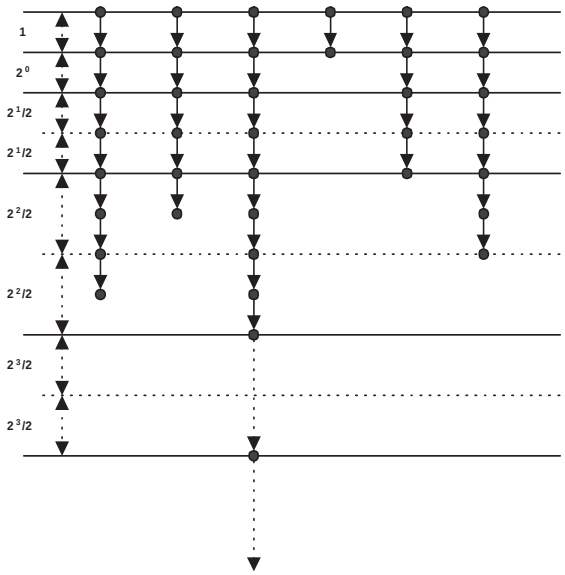


Figure 4: Segmentation of leaf chains and creation of sub-bands in a band.

Each shortest-path tree $T(x)$ has $O(\log^2 n)$ sub-bands of independent paths and for convenience we will number them from top to bottom with consecutive integers starting from 0. By $\beta_{T(x),i}$ we denote the set of paths in the $i^{th}$ sub-band of $T(x)$. In order to compute $dl[x, y, i]$ for each vertex $x$ and for each of the $O(\log^2 n)$ values of $i$, we call *exclude-d*$(G, x, \beta_{T(x),i})$ once. Similarly, for each vertex $y$ and for each of the $O(\log^2 n)$ values of $i$, we compute $dr[x, y, i]$ by calling *exclude-d*$(\hat{G}, y, \beta_{\hat{T}(y),i})$ once.

It is also clear that any shortest path $\pi_{x,y}$ in $T(x)$ can have $O(\log^2 n)$ segments. For convenience we number these segments from $x$ to $y$ with consecutive integers starting from 0.

If a vertex $u$ on $\pi_{x,y}$ lies on the boundary of two consecutive segments in $T(x)$ we assume that the lower numbered segment contains $u$. Any edge $(u, v)$ on $\pi_{x,y}$ is assumed to be contained in the same segment that contains vertex $u$.

Once the paths are segmented, it is straightforward to compute the entries in the matrices *bcl*, *bcr*, *vl* and *vr*.

### Query

The query algorithm is as follows:

```
function distance(x, y, u, v):  ℜ
1.   if d_{x,u} + w_{u,v} + d_{v,y} > d_{x,y} then return d_{x,y} fi
2.   i := bcl[x, u], d := dl[x, y, i]
3.   l := vl[x, y, i], r := vl[x, y, i + 1]
4.   while h_{l,r} > 1 do
5.      if h_{l,v} ≤ ⌈h_{l,r}/2⌉ then
6.         i := bcr[v, r], d := min(d, dr[x, r, i] + d_{r,y})
7.         t := vr[x, r, i + 1], r := vr[x, r, i]
8.         if h_{x,l} > h_{x,t} then l := t fi
9.      else
10.        i := bcl[l, u], d := min(d, d_{x,l} + dl[l, y, i])
```

11. $t := vl[l, y, i+1]$, $l := vl[l, y, i]$
12. **if** $h_{r,y} > h_{t,y}$ **then** $r := t$ **fi**
13. **fi**
14. **end while**
15. **return** $d$

**Correctness**

In line 1, we get rid of the case when $(u, v) \notin \pi_{x,y}$ and return $d_{x,y}$ as the answer. If $h_{x,l} \leq 1$ before entering the *while* loop, $d$ trivially covers $[x, u] \times [v, y]$ and the answer is returned without executing the loop. However, if $h_{x,l} > 1$ , we have $h_{l,r} \leq \frac{h_{x,l}}{2} < \frac{h_{x,y}}{2}$. $[l, r]$ contains $(u, v)$ and $d$ covers $[x, l] \times [r, y]$.

Now, consider what happens in an iteration of the *while* loop. Let us examine the case when $h_{l,v} \leq \lceil \frac{h_{l,r}}{2} \rceil$. Let $p = vr[x, r, i+1]$, $q = vr[x, r, i]$ and $l'$, $r'$ respectively be the new values of $l$, $r$ in the next iteration.

After the assignment in line 9, $d$ covers $[x, p] \times [q, r] \cup [x, l] \times [r, y]$.

If $h_{x,p} < h_{x,l}$ then $l' = p$ and $r' = q$. So, $h_{l',r'} = h_{p,q} \leq \frac{h_{q,r}}{2} < \frac{h_{l,r}}{2} < \frac{3}{4} h_{l,r}$

If $h_{x,p} \geq h_{x,l}$ then $l' = l$ and $r' = q$. Let $m$ be a vertex on $\pi_{l,r}$ such that $h_{l,m} = \lceil \frac{h_{l,r}}{2} \rceil$.

Now, if $h_{l,q} \leq h_{l,m}$, then $h_{l',r'} = h_{l,q} \leq h_{l,m} \leq \lceil \frac{h_{l,r}}{2} \rceil < \frac{3}{4} h_{l,r}$.

Otherwise,

$$\begin{aligned} h_{l',r'} &= h_{l,q} \\ &= h_{l,p} + h_{p,q} \\ &\leq \frac{h_{l,r}}{2} + h_{p,q} && [h_{l,p} \leq h_{l,u} < h_{u,r} \Rightarrow h_{l,p} \leq \tfrac{h_{l,r}}{2}] \\ &\leq \frac{h_{l,r}}{2} + \frac{h_{q,r}}{2} \\ &< \frac{h_{l,r}}{2} + \frac{h_{l,r}}{4} && [h_{l,q} > h_{l,m} \Rightarrow h_{q,r} < h_{m,r} \Rightarrow h_{q,r} < \tfrac{h_{l,r}}{2}] \\ &= \frac{3}{4} h_{l,r} \end{aligned}$$

So, when $h_{l,v} \leq \lceil \frac{h_{l,r}}{2} \rceil$, we always have $h_{l',r'} < \frac{3}{4} h_{l,r}$. Moreover in each of the above cases $[x, l'] \times [r', y] \subseteq [x, p] \times [q, r] \cup [x, l] \times [r, y]$, i.e., $d$ covers $[x, l'] \times [r', y]$. Using similar arguments we can prove the same results for $h_{l,v} > \lceil \frac{h_{l,r}}{2} \rceil$, too.

Note that $[l, r]$ always contains $(u, v)$ and $h_{l,r}$ is reduced by a factor of at least $\frac{4}{3}$ in each iteration of the *while* loop until it becomes equal to 1. So, at the end of the algorithm $[l, r] = [u, v]$. Therefore, $d$ covers $[x, u] \times [v, y]$ eventually. Hence we obtain

CLAIM 6.2. *In algorithm CR-2, the query function distance(x, y, u, v) correctly computes the shortest distance from vertex x to vertex y in G avoiding the edge* $(u, v)$.

**Preprocessing, Query and Space Bounds**

CLAIM 6.3. *In algorithm CR-2, preprocessing requires* $O(mn \log^2 n + n^2 \log^3 n)$ *worst-case time, any distance query can be answered in* $O(\log n)$ *worst-case time, and the data structure requires* $O(n^2 \log^2 n)$ *space.*

*Proof.* In order to initialize the matrices $dl$ and $dr$ the function *exclude-d* is called a total of $O(n \log^2 n)$ times and *exclude-d* runs in $O(m + n \log n)$. It is trivial to see that the cost of computing $dl$ and $dr$ dominates the preprocessing cost. Therefore, preprocessing requires $O(mn \log^2 n + n^2 \log^3 n)$ worst-case time.

The $O(\log n)$ worst-case query time follows from the fact that $h_{l,r} < n$ initially, and is reduced by a

factor of at least $\frac{4}{3}$ in each iteration of the while loop in the query algorithm.

Each of the matrices $d$, $h$, $bcl$ and $bcr$ has size $O(n^2)$, and each of the matrices $vl$, $vr$, $dl$ and $dr$ has size $O(n^2 \log^2 n)$. Therefore, the total space requirement is $O(n^2 \log^2 n)$. □

## 7 Further Extensions and Concluding Remarks

We have presented two improved distance sensitivity oracles for directed graphs with real edge weights, that answer queries asking for the shortest distance from any given vertex to another one avoiding a failed link. The query algorithms can be easily modified to handle the following situations without increasing the query time:

**Arbitrary change in a link weight.** If the weight of a link $(u, v)$ changes from $w_{u,v}$ to $w'_{u,v}$ any distance query asking for the shortest distance from any vertex $x$ to another vertex $y$ in the changed graph can be answered as min(distance($x$, $y$, $u$, $v$), $d_{x,u} + w'_{u,v} + d_{v,y}$).

**At most one link fails and a constant number of links are restored.** If a link $(u, v)$ fails and another link $(u_1, v_1)$ gets restored then the shortest distance from a vertex $x$ to another vertex $y$ can be answered as min(distance($x$, $y$, $u$, $v$), distance($x$, $u_1$, $u$, $v$) + $w'_{u_1,v_1}$ + distance($v_1$, $y$, $u$, $v$)) where $w'_{u_1,v_1}$ is the weight of the restored link $(u_1, v_1)$. If no more than a constant number of links are restored we will need to consider only a constant number of permutations of the restored links along the shortest $x$ to $y$ path in the changed graph. So the query time increases by no more than a constant factor.

There are a number of avenues for further research in this area. Designing efficient oracles that are able to answer distance queries when several links fail at the same time is one direction for further research. Designing more efficient preprocessing algorithms, for instance by using incremental calculations, is another avenue.

It may be possible to obtain more efficient oracles for special types of graphs such as undirected graphs or unweighted graphs. For instance, for undirected graphs the computation may be simplified by avoiding the use of the graph $\hat{G}$. Further, for sparse undirected graphs, the preprocessing time can be improved by using the undirected shortest path algorithm in [PR02] if the ratio $r$ of the maximum to minimum edge weight is not too large (i.e., $r \in 2^{n^{o(1)}}$). By using the algorithm in [PR02] in place of Dijkstra's algorithm the preprocessing time is improved to $O(mn^2\alpha(m, n) \log n + n^3 \log n \log\log r)$ for CR-1 and $O(mn\alpha(m, n) \log^2 n + n^2 \log^2 n \log\log r)$ for CR-2.

## References


[DI01] C. Demetrescu, 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, Navada*, pp. 838-843, 2001.

[DT02] C. Demetrescu, M. Thorup. Oracles for distances avoiding a link-failure. In *Proc. of the 13th IEEE Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02), San Fransisco, California*, pp. 838-843, 2002.

[FT87] M. L. Fredman, R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596-615, 1987.

[KS99] V. King, G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proc. of the 31st ACM Symposium on Theory of Computing (STOC'99)*, pp. 492-498, 1999.

[KT01] V. King, M. Thorup. A space saving trick for directed fully dynamic transitive closure and shortest path algorithms. In *Proc. of the 7th Annual International computing and Combinatorics Conference (COCOON), LNCS 2108*, pp. 268-277, 2001.

[K99] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. of the 40th IEEE Annual Symposium on Foundations of Computer Science (FOCS'99)*, pp. 81-99, 1999.

[NNS89] J. Naor, M. Naor, A. A. Schaffer, "Fast parallel algorithms for chordal graphs," *SIAM J. Computing*, Vol. 18, pp. 327-349, 1989.

[PR02] S. Pettie, V. Ramachandran. Computing shortest paths with comparisons and additions (extended abstract). In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 267-276, 2002.

[R87] V. Ramachandran. Parallel algorithms for reducible flow graphs. *Journal of Algorithms*, 23:1-31, 1997. (Preliminary version in *Princeton Workshop on Algorithm, Architecture, and and Technology*, S.K. Tewksbury, B.W. Dickinson, S.C. Schwartz, ed., 1987, pp. 117-138. Plenum Press.)

[T01] M. Thorup. Fortifying OSPF/IS-IS against link-failure, 2001. Manuscript.