Approximate Single Source Dual Fault Tolerant Distance Oracle

Koustav Das
Department of Computer Science and Engineering, IIT Gandhinagar, India

Manoj Gupta
Department of Computer Science and Engineering, IIT Gandhinagar, India

Abstract

We are given an undirected weighted graph G with n vertices and m edges, edge weights in $[1, W]$, and a designated source vertex s. We design a single source dual fault tolerant distance oracle for G. Given a destination vertex t and a set F of at most two faulty edges, the oracle returns a $(1 + O(\varepsilon))$-approximation of the weight of the shortest path from the source s to t avoiding F. Our oracle uses $\tilde{O}(n\sqrt{n})$ space[1] and has $\tilde{O}(1)$ query time.

Prior to our result, single source *single* fault tolerant oracles were known to return a $(1 + \varepsilon)$ approximation of the weight of the shortest path using $\tilde{O}(n)$ space and $O(1)$ query time. However, extending these approaches to multiple faults remained an open problem. Indeed, all $(1 + \varepsilon)$-approximate distance oracles that handle multiple faults require $\Omega(n^2)$ space. We break this bound by presenting the first dual fault tolerant distance oracle with $o(n^2)$ space.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Shortest paths; Theory of computation → Data structures design and analysis

Keywords and phrases Fault-tolerant distance oracle, Shortest paths, Graph algorithms, Dual failures, Approximate distance oracle

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.90

Related Version *Full Version*: `https://arxiv.org/pdf/2607.02999` [10]

Funding *Manoj Gupta*: Supported by ANRF grant ANRF/ARGM/2025/001853/MTR

1 Introduction

In many real-world scenarios, we are interested in computing the shortest distances from a specific location to all other reachable locations in a network. Such networks can be modelled as graphs, where vertices represent places and edges capture connectivity between them. This naturally leads to the classical *single source shortest paths* (SSSP) problem, where the objective is to preprocess a graph so that distances from a designated source s to any other vertex can be efficiently retrieved in response to queries.

Query(s, t) : Find the weight of the shortest path from s to t.

For unweighted graphs, the single source shortest paths (SSSP) can be efficiently computed using a breadth-first search (BFS). The distances from the source to all other vertices can be stored in $O(n)$ space, allowing $O(1)$ query time per vertex, where n and m denote the number of vertices and edges, respectively. For weighted graphs, classical SSSP algorithms such as Dijkstra's algorithm allow us to preprocess the graph in $O(m + n \log n)$ time so that distances from a designated source can be queried in constant time. Over the years, a rich body of work has advanced the classical SSSP problem [16, 15, 23, 21, 22, 18]. Later, Duan, Mao, Shu, and Yin [12] developed a randomized algorithm for the SSSP problem

[1] poly$(\log_{1+\varepsilon} nW)$ factors are hidden in the $\tilde{O}$ notation.

with a time complexity of $O(m\sqrt{\log n \log\log n})$. Recently, Duan, Mao, Mao, Shu, Yin [11] achieved further progress by presenting a deterministic $O(m \log^{2/3} n)$-time algorithm for directed graphs with non-negative edge weights.

However, in practice, some connections may fail or become temporarily unavailable. We model these disruptions as *faulty edges*, i.e., edges removed from the graph. The task is to determine the length of the shortest path from a specified source vertex s to any other vertex $t \in V$ while avoiding faulty edges. Our objective is to answer the following query efficiently:

$\textsc{Query}(t, F)$: Find the weight of the shortest path from s to t avoiding all the edges contained in F

An algorithm may preprocess the graph G and create a data structure to quickly answer the above query. Such an algorithm (and the data structure) is called a *distance oracle* in literature. Since we are dealing with a single source s and must output shortest paths avoiding faults, our oracle is called *Single Source fault tolerant* distance oracle.

Let us first see a simple single source fault tolerant distance oracle. One could naively precompute and store exact shortest paths from the source s to all other vertices under all possible faulty edge scenarios. However, this approach is computationally prohibitive, as both the preprocessing time and space taken by the data structure grow rapidly with the number of faults. For example, even with only two faulty edges, the space requirement already becomes $O(m^2 n)$. Another naive approach is not to preprocess the graph at all. Upon $\textsc{Query}(t, F)$, we run Dijkstra's algorithm from s after removing the faulty edges from the graph. This distance oracle is space-efficient, but the query time is too high.

In our paper, we focus primarily on the single source fault tolerant distance oracle. However, in the literature, a lot of work has also been done on the all-pairs variant, where any vertex can serve as the source. To distinguish between these variants, we use the following notation:

- $\textsc{Sdo}(f)$: A single source fault tolerant distance oracle handling up to f faults.
- $\textsc{Do}(f)$: An all-pairs fault tolerant distance oracle handling up to f faults.

Distance oracles are further classified by the quality of their output. We say an $\textsc{Sdo}(f)$ is **α-approximate** if, for all $t \in V$ and all valid fault sets F ($|F| \leq f$), the query returns a value that is at least the length of the shortest path from s to t avoiding F, and at most α times this length. If $\alpha = 1$, the oracle[2] is **exact**. These definitions naturally extend to $\textsc{Do}(f)$.

We now summarise key results on single source and (general) distance oracles under failures in undirected graphs. Gupta and Singh [17] designed an $\textsc{Sdo}(1)$ of size $\tilde{O}(n^{3/2})$ that answers queries in $\tilde{O}(1)$ time for an undirected unweighted graph. Later, Bilò, Cohen, Friedrich, and Schirneck [8] extended this result to weighted graphs with an oracle size of $O(W^{1/2} n^{3/2})$, where W is the maximum edge weight. Next, we discuss results on approximate distance oracles.

Baswana and Khanna [3] initiated the systematic study of approximate fault tolerant distance oracles. They developed an $O(m \log n)$-time constructible $\textsc{Sdo}(1)$ of size $O(n \log n)$ reporting a 3-approximate shortest-path distance from a fixed source to any vertex $v \in V$ while avoiding any failed vertex $x \in V$. For undirected, unweighted graphs, they also proposed an $O\left(\frac{n \log n}{\epsilon^3}\right)$-space $\textsc{Sdo}(1)$ that reports $(1+\epsilon)$-approximate shortest paths for any $\epsilon > 0$. More recently, [2] constructed an $\textsc{Sdo}(1)$ of size $O\left(n \log_{1+\epsilon}(nW)\right)$ with query time $O\left(\log\log_{1+\epsilon}(nW)\right)$ for directed weighted graphs.

[2] For exact distance oracle, $\tilde{O}$ notation hides poly $\log n$ factor.

One may ask if there are any result on exact or $(1+\varepsilon)$-approximate SDO(f) for $f \geq 2$. Yes, there are, but only indirectly. To this end, notice that a DO(f) is also an SDO(f). We now show some relevant results related to DO(f) where $f \geq 2$. Duan and Pettie [13] designed an exact DO(2) of size $\tilde{O}(n^2)$ with $\tilde{O}(1)$ query time. Chechik, Cohen, Fiat, Kaplan [9] designed a $(1+\epsilon)$-approximate DO(f) that tolerates up to $f = O\left(\frac{\log n}{\log\log n}\right)$ faults with $\tilde{O}(n^2)$ space and $O(f^5)$ query time. Both these algorithms also imply the existence of exact and $(1+\varepsilon)$-approximate SDO(f), though requiring $\tilde{O}(n^2)$ space. One of the main questions in this field is to design an SDO(2) (and in general SDO(f) for $f \geq 2$) with sub-quadratic space.

In this work, we narrow this gap by studying the single source setting under up to two edge failures. To the best of our knowledge, we provide the first approximate SDO(2) that requires only $o(n^2)$ space while supporting queries in nearly constant time.

Our Result

▶ **Theorem 1.** *Given an undirected graph G with edge weights in $[1, W]$, there is a $(1+O(\varepsilon))$ approximate* SDO(2) *of size $O(n\sqrt{n}\log^4_{1+\varepsilon} nW)$ and $O(\log^c_{1+\varepsilon} nW)$ query time where $c > 0$ is a constant.*

2 Preliminaries

Let $G = (V, E)$ be an undirected weighted graph with n vertices and m edges with edge weights in $[1 \dots W]$. For any $s, t \in V$, st denote a shortest path from s to t, and $|st|$ its weight. For any path P, $|P|$ denotes the weight of the path P. Let $P[a, b]$ denotes the subpath from a to b in P. Given two paths P_1 and P_2 such that P_1 ends where P_2 begins, their concatenation is denoted $P_1 \odot P_2$. Let $st \diamond F$ denotes the shortest path from s to t that avoids all edges in F, and $st \diamond e$ when $F = \{e\}$ for any edge $e = (u, v)$. All edges in F are called *faulty* or *failed* edges. All vertices incident to edges in F, that is, the vertices in the set $V(F)$ are called *affected* vertices. Given a path $P[s, t]$, we normally visualize it as drawn on a plane with s at the top and t at the bottom. For an edge $e = (u, v) \in P$ (where u is closer to s), the term *above e on path P* refers to the subpath $P[s, u]$, and *below e on path P* refers to the subpath $P[v, t]$. Similarly, we can define the notion of *above/below of a vertex v* on a path P. There can be many faulty edges on the path P. The faulty edge closest to t on the path P is called the *last faulty* edge on P.

We assume that the shortest path between any pair of vertices in the graph is unique, even in the presence of edge failures. This can be ensured by applying a standard weight-perturbation technique in which each edge is assigned an infinitesimally small distinct weight (e.g., see [20]). As a result, any two distinct st paths will have different total weights. Similar assumptions have been used in prior works such as [5, 19, 17, 14].

▶ **Definition 2** (k-decomposable paths). *A path is* k-decomposable *if it is the concatenation of at most $k+1$ shortest paths interleaved with at most k edges, where $k \geq 0$. For example, $P_1 \odot e \odot P_2$ is a 1-decomposable path if both P_1 and P_2 are shortest paths in the graph G.*

The following lemma formalises the importance of decomposable paths.

▶ **Lemma 3** ([1]). *After k edge failures in a weighted graph, each new shortest path is k-decomposable.*

When $|F| = 2$, by the above lemma, the path $st \diamond F$ can be written as the concatenation of at most three shortest paths interleaved with at most two edges. We call such a path *2-decomposable*. Similarly, when $|F| = 1$, the path $st \diamond F$ can be expressed as the concatenation of at most two shortest paths interleaved with at most one edge, referred to as a *1-decomposable* path. We now redefine some notation from [9].

▶ **Definition 4** (Netpoints). *Let P be a (not necessarily shortest) path from s to t in G. We define the set $L = \{u_0, u_1, \dots\}$ such that u_i is the first vertex on P satisfying:*

$|P[s, u_i]| \geq (1+\varepsilon)^i$.

If no such vertex exists for a given i, the sequence stops. Similarly, define $R = \{v_0, v_1, \dots\}$ as the set of vertices on P where v_i is the first vertex (traversing from t to s) such that:

$|P[v_i, t]| \geq (1+\varepsilon)^i$.

We define the netpoints *of P as the set $L \cup R \cup \{s, t\}$. The number of netpoints is $O(\log_{1+\varepsilon} |P|) = O(\log_{1+\varepsilon} nW)$.*

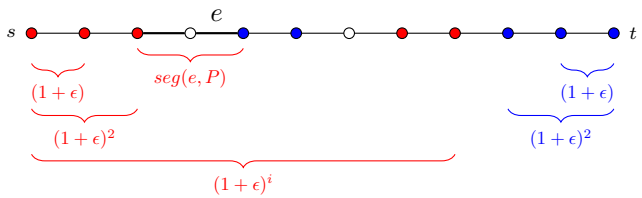


Figure 1 Netpoints originating from the vertex s are shown in red and form the set L. Netpoints originating from the vertex t are shown in blue and form the set R. A vertex may belong to both L and R. The union of these sets constitutes the set of netpoints.

▶ **Definition 5** (Segment). *A* segment *of a path P is any subpath between two consecutive netpoints on P. For an edge $e \in P$, define $seg(e, P)$ as the unique segment in P that contains e.*

In [9], the authors proved the following lemma that bounds the length of the segment containing an edge e.

We state the lemma below and, for the sake of completeness, prove it in the appendix (See Appendix A)

▶ **Lemma 6** ([9], Lemma 2.1). *Let P be a path from s to t, and let $e = (u, v)$ be an edge of P (where u is closer to s than t on P). Then either* $\text{seg}(e, P) = \{e\}$ *or* $|\text{seg}(e, P)| \leq \varepsilon \cdot \min\{|P[s, u]|, |P[u, t]|\} \leq \varepsilon |P|$.

The concept of a detour frequently appears in fault tolerant shortest path literature. Normally, detours are defined for paths that avoid some edges. In our paper, we will define detour for paths that have one extra property. We begin by defining the traditional notion of a detour, starting with the following definition.

▶ **Definition 7** (Prefix and Suffix of path P). *Let P be a (not necessarily shortest) path from s to t. Let $e = (u, v)$ be an edge on P, where u is closer to s and v is closer to t. Then $\textsc{Prefix}(P, e)$ is the subpath $P[s, u]$ and $\textsc{Suffix}(P, e)$ is the subpath $P[v, t]$.*

We now define the detour of a path avoiding a single edge and the start of the detour.

▶ **Definition 8.** *(Detour of a path R) Let R be a path from s to t that avoids an edge e on the original shortest path st. Let w be the last vertex of R that also lies on $\textsc{Prefix}(st, e)$, that is, the path $R[w, t]$ does not intersect with $\textsc{Prefix}(st, e)$ except at w. Then the subpath $R[w, t]$ is the* detour *of R, and we say the detour of R* starts *at w.*

Now, we define special detours that follow some other properties.

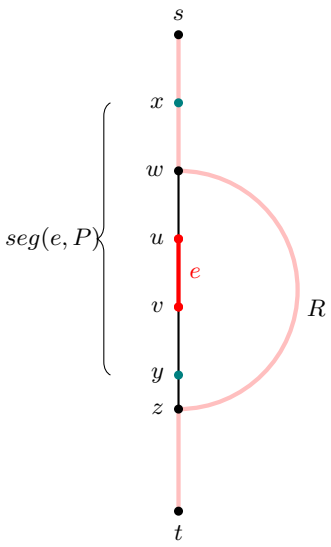


Figure 2 Let P be a path from s to t. Let R be a path (shown in pink) avoiding e that lies on the segment xy. Detour of R is $R[w,t]$. The detour of R starts on the segment xy above e at w and joins the path at z below the segment xy.

▶ **Definition 9** (Detour of a path R starting from a segment and joining below it)**.** *Let R be a path avoiding an edge e on the st path. Let w be the last vertex of R on $\textsc{Prefix}(st, e)$. We say that the detour of R starts from w on the segment $seg(e, st)$ if:*

1. *$w \in seg(e, st)$, and*
2. *$R[w,t]$ does not intersect $seg(e, st)$ except at point w. (See Figure 2)*

Let z be the first vertex of R such that $z \in \textsc{Suffix}(st, e)$, that is the path $R[s,z]$ does not intersect with $\textsc{Suffix}(st, e)$ except at z. By the second condition above, z does not lie on the segment $seg(e, st)$ and we say that the detour joins st below $seg(e, st)$ at z.

3 Overview

Consider the algorithm of [9], which designs a $(1+\varepsilon)$-approximate $\textsc{Do}(f)$ with size roughly $\tilde{O}(n^2)$ and query time $O(f^5)$. We focus on the oracle's space complexity. The algorithm builds a data structure $\textsc{Ft}(st)$ for every vertex pair (s,t) in the graph. While computing the shortest s-t path avoiding F, we query $\textsc{Ft}(st)$. This structure guarantees the following property.

▶ **Lemma 10** (stated informally, omitting technical details)**.** *Either $\textsc{Ft}(st)$ outputs $|st \diamond F|$, or there exists an affected vertex $w \in V(F)$* **close** *to the path $st \diamond F$. In the latter case, we can compute $|sw \diamond F|$ exactly using $\textsc{Ft}(sw)$.*

Note that the above lemma is only an informal summary of [9] and omits technical details for clarity. The lemma states that using $\textsc{Ft}(st)$, we either obtain $|st \diamond F|$ directly or identify an affected vertex $w \in V(F)$ **close** to the path $st \diamond F$ (the notion of closeness is not formally defined here). We then compute $|sw \diamond F|$ and $|wt \diamond F|$. Since w is close to $st \diamond F$, their sum approximates $|st \diamond F|$ well. The lemma ensures that $\textsc{Ft}(sw)$ returns $|sw \diamond F|$ exactly. For $|wt \diamond F|$, the source is now an affected vertex, so we recurse and compute it using $\textsc{Ft}(wt)$. The recursion depth is bounded by $2f$, the number of vertices in $V(F)$. In fact, [9] implement this process in $O(f^5)$ time.

We now focus on the space taken by the data-structure. In [9], the authors constructed an $(1+\varepsilon)$-approximate $\textsc{Do}(f)$ for all-pairs distance queries, which justified their $\tilde{O}(n^2)$ space. In contrast, our setting involves only a single source, where we aim to design a $\textsc{Sdo}(2)$ of size $o(n^2)$ with fast query time. Thus, we cannot construct an $\textsc{Ft}$ data structure for every vertex pair in the graph. At the same time, we require fast query time, and achieving both objectives simultaneously is challenging. To highlight this difficulty, we next examine a related problem.

In [6] (and then in a subsequent improvement in [7]), the authors design a $(3+\varepsilon)$-approximate $\textsc{Do}(f)$. Their oracle has size $o(n^2)$ but suffers from $\Omega(n^c)$ query time (for some constant $c<1$). They face the same challenge as ours: using the $\textsc{Ft}$ data structure requires quadratic space if built for all pairs. To avoid this, they sample a subset of vertices and construct $\textsc{Ft}$ data structures from each sampled vertex to all others. Since the sample size is much smaller than n, the overall space remains $o(n^2)$. However, this space-saving comes at the cost of $O(n^c)$ query time. Reducing this query time remains an important open problem.

Our challenge closely resembles the above problem, even though we focus on the simpler single source setting. In fact, our goal is harder, as we seek polylogarithmic query time. The reader will observe that we introduce significant modifications to the $\textsc{Ft}$ data structure, enabling us to reduce the number of vertex pairs for which it needs to be constructed. To fully appreciate our new contribution, we first present a construction of $\textsc{Sdo}(1)$ that achieves $\tilde{O}(n\sqrt{n})$ space and $\tilde{O}(1)$ query time. Although this result is weaker than that of [3], who obtained nearly $\tilde{O}\left(\frac{n}{\varepsilon^3}\right)$ space and $O(1)$ query time, it serves as an essential foundation for our improved dual fault structure.

Our construction naturally extends to handle the case of two faults. Once the reader is familiar with the working of $\textsc{Sdo}(1)$, the extension to $\textsc{Sdo}(2)$ becomes straightforward. Thus, the next section acts as an overview of our approach.

4 Warm-up: Single fault tolerant Distance Oracle

In this section, we design a single fault tolerant distance oracle. The space taken by our distance oracle is $\tilde{O}(n\sqrt{n})$ and its query time is $\tilde{O}(1)$. Let us first see the construction of the data-structure that will be used by this oracle.

4.1 Constructing Ft(st)

We construct a data-structure $\textsc{Ft}(st)$ for each $t \in V$. We build $\textsc{Ft}(st)$ using the function $\textsc{MakeNode}$ defined in Algorithm 1. This function takes three parameters: a graph H, a path P, and the *depth* of the node. It creates a node with these inputs and recursively builds its subtree. To construct the full tree $\textsc{Ft}(st)$, we invoke $\textsc{MakeNode}(G, st, 0)$, which initializes the root node at level 0. Since Algorithm 1 only recurses when depth < 1, the tree has exactly two levels: the root at depth 0 and its children at depth 1.

In $\textsc{MakeNode}(G, st, 0)$, we first make the root of $\textsc{Ft}(st)$, that is $\textsc{Root}(\textsc{Ft}(st))$. Each node of $\textsc{Ft}(st)$ is associated with a graph and a primary path $\mathbf{P}_{\mathcal{R}}$. For instance, for the root $\mathcal{R}$ of $\textsc{Ft}(st)$, the associated graph is G, and the primary path $\mathbf{P}_{\mathcal{R}}$ is the st path. For each segment $xy \in \mathbf{P}_{\mathcal{R}}$, we do the following:

1. Make a Segment child
   Let $P(xy)$ be the shortest 1-decomposable path of the form $Q_1 \odot e_1 \odot Q_2$ in the original graph G, where each Q_j is a shortest path in the original graph, such that no edge of $P(xy)$ belongs to the segment xy. In other words, $P(xy)$ is a 1-decomposable path in G that survives in $G - xy$. If $P(xy)$ exists, then we create a child for the segment xy by calling $\textsc{MakeNode}(G - xy, P(xy), 1)$ and designate it as a *segment* child.

Algorithm 1 MakeNode$(H, P, depth)$.

```
1  Make a node R with path P_R = P
2  if depth < 1 then
3      foreach segment xy in P_R do
4          Let P(xy) be the shortest 1-decomposable path Q_1 ⊙ e_1 ⊙ Q_2 in the original
             graph G, where each Q_j is a shortest path in G, such that no edge of P(xy)
             belongs to the segment xy
5          if P(xy) exists then
6              a child of R ← MakeNode(H − xy, P(xy), depth + 1)
7          foreach i = 0 to log_{1+ε} nW do
8              Let Q = {P_R ⋄ e | e ∈ xy} // P_R ⋄ e denotes the shortest path
                 between the endpoints of P_R avoiding edge e, computed in
                 G
9              Let P_i be the path in Q whose detour starts closest to x on the segment
                 xy and detour joins below xy on P_R, and its detour length ≤ (1 + ε)^i
10             if P_i exists then
11                 a child of R ← MakeNode(H, P_i, depth + 1)
12 return R
```

2. Make Detour children
Let $\mathcal{Q}$ be the set of all shortest paths between the endpoints of $\mathbf{P}_{\mathcal{R}}$ that avoid some edge of the segment xy, computed in G. Formally, $\mathcal{Q} = \{\mathbf{P}_{\mathcal{R}} \diamond e \mid e \in xy\}$. Let P_i be the path in $\mathcal{Q}$ such that the detour starts closest to x on the segment xy, the detour joins $\mathbf{P}_{\mathcal{R}}$ below xy on $\mathbf{P}_{\mathcal{R}}$, and the detour length is $\leq (1+\varepsilon)^i$. If P_i exists, then we create a *detour* child $\mathcal{R}_i$ of $\mathcal{R}$ with P_i as its primary path by invoking MakeNode$(G, P_i, 1)$.

For each segment xy, we create one segment child and at most $\log_{1+\varepsilon} nW$ detour children of $\mathcal{R}$. Since $\mathbf{P}_{\mathcal{R}}$ contains $O(\log_{1+\varepsilon} nW)$ segments, there are $O(\log_{1+\varepsilon} nW)$ segment children and $O(\log^2_{1+\varepsilon} nW)$ detour children at level 1 of FT(st). We thus claim the following:

▶ **Lemma 11.** *There are $O(\log^2_{1+\varepsilon} nW)$ nodes in FT(st).*

We now try to bound the size of FT(s,t). To this end, we claim the following crucial lemma.

We state the following lemma. For the proof, see the full version of the paper [10].

▶ **Lemma 12.** *For each node $\mathcal{X} \in$ FT(st), $\mathbf{P}_{\mathcal{X}}$ is 1-decomposable.*

We now use the above lemma to store the primary path at each node of FT(st) efficiently. Consider a node $\mathcal{R}$ of FT(st). The lemma implies that $\mathbf{P}_{\mathcal{R}}$ is 1 decomposable, so it has the form $Q_1 \odot e \odot Q_2$. If a path Q_j has at most $\sqrt{n}$ edges, we store it explicitly at node $\mathcal{R}$. If Q_j has more than $\sqrt{n}$ edges, we apply a sampling approach to represent Q_j. A similar technique has been used in [6]. We sample a set L of nodes from G independently with probability $\tilde{O}\left(\frac{1}{\sqrt{n}}\right)$. With high probability, $|L| = \tilde{O}(\sqrt{n})$. We store the shortest path tree rooted at each node in L, which requires total space $\tilde{O}(n\sqrt{n})$. Now consider a subpath Q_j from a to b. Since Q_j has $\geq \sqrt{n}$ edges, a vertex $z \in L$ lies on Q_j with high probability. Moreover, since subpaths of shortest paths are themselves shortest paths, both $Q_j[a,z]$ and $Q_j[z,b]$ are

shortest paths in G. We therefore store Q_j implicitly by recording only the pairs (a, z) and (z, b) at node $\mathcal{R}$, and recover the full paths $Q_j[a, z]$ and $Q_j[z, b]$ from the shortest path tree T_z rooted at $z \in L$. Thus, each node $\mathcal{R}$ stores Q_j in $O(1)$ space when Q_j has $\geq \sqrt{n}$ edges.

Combining both cases: each node $\mathcal{R}$ represents its primary path in $O(\sqrt{n})$ space (either the explicit path of length at most $\sqrt{n}$, or an $O(1)$-sized pairs (a, z) and (z, b)). Since $\textsc{Ft}(st)$ has $O(\log^2_{1+\varepsilon} nW)$ nodes by Lemma 11, each tree occupies $\tilde{O}(\sqrt{n})$ space. Across all n trees, the total space is $\tilde{O}(n\sqrt{n})$. Also, we build shortest path trees from all vertices of L. This also takes $\tilde{O}(n\sqrt{n})$ space. Thus, the total space taken by our data-structure is $\tilde{O}(n\sqrt{n})$.

In our query algorithm, we must efficiently check whether a given edge e lies on the path Q_j. If Q_j has $\leq \sqrt{n}$ edges, we store a dictionary at node $\mathcal{R}$ using $\tilde{O}(\sqrt{n})$ space, allowing constant-time membership queries for $e \in Q_j$. If Q_j has $> \sqrt{n}$ edges, we represent Q_j implicitly as the concatenation of two shortest paths, $Q_j[a, z]$ and $Q_j[z, b]$, where $z \in L$ lies on Q_j. Now we use the following lemma to check if e lies in $Q_j[a, z]$.

▶ **Lemma 13** (See [4] and its references)**.** *Given a tree T with n vertices, we can construct a data structure of size $O(n)$ in $O(n)$ time, allowing us to answer LCA(Lowest Common Ancestor) queries in $O(1)$ time.*

As we are storing the path $Q_j[a, b]$ as a pairs (a, z) and (z, b) where $z \in L$, we will proceed as follows. For a faulty edge $e = (u, v)$, we first check if T_z contains e. If T_z does not contain e, then Q_j cannot contain e. If T_z contains e, then we check whether $LCA(a, u) = u$ and $LCA(a, v) = v$ are satisfied or not. If both conditions hold, then the edge e is in the shortest path az. These LCA queries take $O(1)$ time. A similar check applies for zb. With an additional $\tilde{O}(1)$ data structure at $\mathcal{R}$, we can also report which segment of Q_j contains e. We prove this later (see Appendix B).

▶ **Lemma 14.** *For each t, each node of $\textsc{Ft}(st)$ can be represented using $\tilde{O}(\sqrt{n})$ space. Thus, the total space taken by $\textsc{Ft}(st)$ over all $t \in V \setminus \{s\}$ is $\tilde{O}(n\sqrt{n})$. Moreover, the time taken to determine if $e \in \mathbf{P}_{\mathcal{R}}$ is $\tilde{O}(1)$ for each node $\mathcal{R}$ in $\textsc{Ft}(st)$. Moreover, we can also report the segment of $\mathbf{P}_{\mathcal{R}}$ that contains e in $\tilde{O}(1)$ time.*

4.2 Query Algorithm

Given a destination vertex t and a faulty edge $e = (u, v)$, we execute the algorithm $\textsc{Query}(t, \textsc{Root}(\textsc{Ft}(st)))$; assuming $F = \{e\}$ is implicitly known. Our algorithm also requires one additional implicit array, which we describe next. For brevity, we omit it from the procedure's arguments.

The procedure Query is recursive. The initial call uses t as the first parameter (the destination), while subsequent calls use one of the affected vertices (see Line 12). The first **if** condition bounds the recursion depth. During $\textsc{Query}(t, \cdot)$, we track the recursion path using an array that records all vertices visited along the current path. The array has constant size because, after the root, each recursive call involves only affected vertices. If Query encounters a vertex that already appears as the first parameter on the current path, it returns ∞.

Let $\mathcal{R}$ be the root of $\textsc{Ft}(st)$. Thus, $\mathbf{P}_{\mathcal{R}} = st$. If st contains no faulty edge, we return it directly. Otherwise, $e = (u, v)$ lies on the st path, and let xy denote the segment of st that contains e. The algorithm then proceeds through three main cases depending on the structure of $st \diamond e$. We now go through these three cases.

Algorithm 2 $\textsc{Query}(t, \mathcal{R})$.

```
1  if any ancestor function call of QUERY had t as the first parameter then
2      return ∞
3  if P_R does not contain e then
4      return |P_R|
5  else
6      PATH ← ∞
7      Let e = (u, v) be the faulty edge on P_R, and let xy ∈ P_R be the segment
         containing e
       ▷ Part 1:  st ⋄ e avoids segment xy
8      Let X be the segment child of R corresponding to the path avoiding xy
9      if X exists then
10         PATH ← |P_X|
       ▷ Part 2:  st ⋄ e intersects segment xy below e
11     if v ≠ t then
12         PATH ← min{PATH, QUERY(v, ROOT(FT(sv))) + |P_R[v, t]|}
       ▷ Part 3:  st ⋄ e intersect segment xy above e
13     foreach i = 0 to log_{1+ε} nW do
14         Let R_i be the detour child of R whose detour in P_{R_i} starts on xy and joins
             below y on P_R and has length at most (1 + ε)^i
15         if R_i exists and P_{R_i} avoids e then
16             PATH ← min{PATH, |P_{R_i}|}
17 return PATH
```

1. Part 1: $st \diamond e$ avoids the segment xy (See part 1 of Figure 3)
   Let $P(xy)$ be the shortest 1-decomposable path in G that survives in $G - xy$ constructed during preprocessing. Let $\mathcal{X}$ be the segment child that contains $P(xy)$. If $\mathcal{X}$ exists, then we set PATH to $|\mathbf{P}_{\mathcal{X}}| = |P(xy)|$.
2. $st \diamond e$ intersect xy.
   There are two cases here, depending on whether $st \diamond e$ intersects xy above or below e. If it intersects xy both above and below e, we will give preference to its intersection below e in the analysis. Let's go over these two cases.
   a. Part 2: $st \diamond e$ intersects xy below e. (See part 2 of Figure 3)
      Let v be the endpoint of e closest to t on the $\mathbf{P}_{\mathcal{R}} = st$ path. We estimate $|st \diamond e|$ by computing $|\mathbf{P}_{\mathcal{R}}[v, t]| + |sv \diamond e|$. Note that for a single fault, the first path is just the shortest path vt. Although $|sv \diamond e|$ is unknown, we can approximate it by calling $\textsc{Query}(v, \textsc{Root}(\textsc{Ft}(sv)))$. One may question whether this recursive call makes any progress. It does - instead of querying at t, we now query at v, an affected vertex, thereby making some progress.
   b. Part 3: $st \diamond e$ intersects xy above e. (See part 3 of Figure 3)
      It means that $st \diamond e$ does not intersect xy below e. This implies that the detour of $st \diamond e$ begins on the segment xy above e and joins below xy on st; a hard case for our algorithm. We had precomputed detour children of the root precisely for such situations, and now we utilise them. For each $i \in [0, \log_{1+\varepsilon} nW]$, we examine the detour child $\mathcal{R}_i$ of $\mathcal{R}$ such that the detour of $\mathcal{R}_i$ starts on xy and has length at most $(1 + \varepsilon)^i$. If $\mathbf{P}_{\mathcal{R}_i}$ avoids e, we update PATH to $|\mathbf{P}_{\mathcal{R}_i}|$.

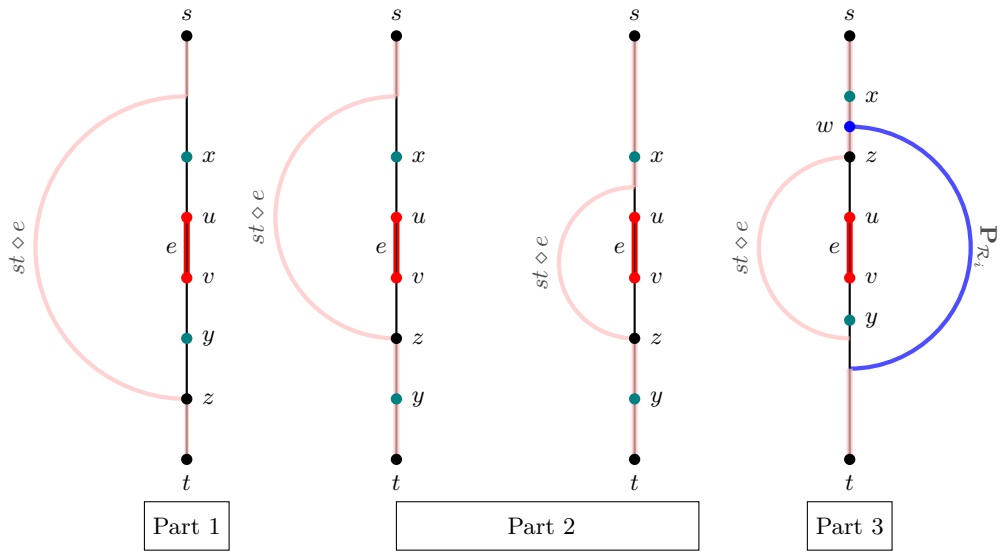


Figure 3 Part 1: $st \diamond e$ avoids segment xy. Part 2: $st \diamond e$ intersects xy below e on segment xy. Part 3: $st \diamond e$ intersects xy above e on segment xy.

4.3 Running Time

In this section, we analyse the running time of our QUERY algorithm. To this end, we introduce a simple method that will also be useful for handling the dual-fault case. We represent the execution of QUERY using a *recursion tree*, where each node corresponds to a call to QUERY. The root is $\text{QUERY}(t, \text{ROOT}(\text{FT}(st)))$, and its children are the recursive calls triggered by this invocation. Recursively expanding the children yields the full tree, which succinctly captures the running time of the algorithm.

The running time of QUERY per node is dominated by the loop in Part 3 (see Line 13). taking $O(\log_{1+\varepsilon} nW)$ time, and it makes at most one recursive call. Consequently, in our case, the recursion tree degenerates to a path. We will show that this path has length at most 3, resulting in a total of 4 nodes. Hence, the overall running time of the algorithm is $O(\log_{1+\varepsilon} nW)$.

In the recursion tree of QUERY, every call after the root uses an affected vertex as its first parameter. Since there are only two affected vertices, any attempt to reuse an affected vertex terminates immediately, returning ∞. Consequently, the path in the recursion tree can have at most three recursive calls. Thus, we can claim the following lemma.

▶ **Lemma 15.** *The running time of* QUERY *is* $O(\log_{1+\varepsilon} nW)$.

We now proceed to prove the correctness of our algorithm. To this end, we first establish a useful property of our algorithm.

4.4 Property of our algorithm

Our algorithm addresses two main challenges: (1) when $st \diamond F$ intersects a segment containing a faulty edge below e, and (2) when the detour of $st \diamond F$ starts above e on the same segment. We first present a simple lemma to handle the first case. This lemma is general-purpose, applying even in the presence of two faults, and will be used extensively in the analysis of the dual fault case.

We state the following lemma. For the proof, see the full version of the paper [10].

▶ **Lemma 16.** *Let $\mathcal{R}$ be a node in $\text{FT}(st)$. Let $e = (u, v)$ be the* **last** *faulty edge on the path $\mathbf{P}_{\mathcal{R}}$ with u closer to s. Let $S = seg(e, \mathbf{P}_{\mathcal{R}})$. Suppose the fault tolerant path $R = st \diamond F$ intersects S at a vertex z below e. Assume there exists a path Q from z to t that also avoids F. If the following conditions hold:*

1. *$\text{QUERY}(v, \text{ROOT}(\text{FT}(sv)))$ returns an α-approximation of $|sv \diamond F|$,*
2. *$|Q|$ is an α-approximation of $|R[z, t]|$, and*
3. *$|S|$ is a β-approximation of $|R|$ where $0 \leq \beta \leq 1$*

then the path of length $|Q| + |S[z, v]| + \text{QUERY}(v, \text{ROOT}(\text{FT}(sv)))$ gives an $(\alpha + \beta + \alpha\beta)$-approximation of $|st \diamond F|$.

We now show how to apply the lemma. Suppose we are processing the root $\mathcal{R}$ of $\text{FT}(st)$, and edge e lies on the segment $S = xy$ of the path $\mathbf{P}_{\mathcal{R}} = st$. Assume that $st \diamond e$ intersects xy below e at the vertex z. In Line 12 we compute

$\text{PATH} = |\mathbf{P}_{\mathcal{R}}[v, t]| + \text{QUERY}(v, \text{ROOT}(\text{FT}(sv)))$

Since z lies between v and t on that path $\mathbf{P}_{\mathcal{R}}$, we get:

$$= |\mathbf{P}_{\mathcal{R}}[z, t]| + |\mathbf{P}_{\mathcal{R}}[z, v]| + \text{QUERY}(v, \text{ROOT}(\text{FT}(sv)))$$

Since $\mathbf{P}_{\mathcal{R}}[z, v]$ lies entirely in the segment S, we have $\mathbf{P}_{\mathcal{R}}[v, z] = S[z, v]$

$$= |\mathbf{P}_{\mathcal{R}}[z, t]| + |S[z, v]| + \text{QUERY}(v, \text{ROOT}(\text{FT}(sv)))$$

Setting $Q = \mathbf{P}_{\mathcal{R}}[z, t]$, we get

$$= |Q| + |S[z, v]| + \text{QUERY}(v, \text{ROOT}(\text{FT}(sv))) \quad (1)$$

Thus, the path computed by our algorithm in Line 12 is the same as the path described in Lemma 16.

4.5 Correctness

Consider finding the shortest path from s to t that avoids the edge $e = (u, v)$. In the function $\text{QUERY}(t, \text{ROOT}(\text{FT}(st)))$, the first parameter is initially t, and in subsequent recursive calls it may be an affected vertex. Order t and the affected vertices by their distance from s in the graph $G - e$, and let L denote the subsequence of all vertices up to t in this order. If t is the closest vertex to s in $G - e$, L contains only t. In the worst case, the size of L is 3. We now prove the following lemma.

▶ **Lemma 17.** *Let p be the k-th vertex in the sequence L where $k \leq |L|$. Then $\text{QUERY}(p, \text{ROOT}(\text{FT}(sp)))$ returns a $(1 + k^3\varepsilon)$-approximation of $|sp \diamond e|$.*

The above lemma implies that $\text{QUERY}(t, \text{ROOT}(\text{FT}(st)))$ returns a $(1+27\varepsilon)$ approximation of $|st \diamond e|$ as $|L| \leq 3$. We prove the lemma by induction on the vertices of L. For the base case ($k = 1$), without loss of generality, let u be the first vertex in the sequence. The shortest su path avoids the faulty edge e, and $\text{QUERY}(u, \text{ROOT}(\text{FT}(su)))$ returns $|su|$ exactly, establishing the base case. For the induction step, assume that there are $k - 1$ vertices before p in L. We now prove it for the k-th vertex p. For notational convenience, we relabel p as t, as this notation aligns with all our lemmas and discussions above.

If the faulty edge e does not lie on the shortest path $\mathbf{P}_{\mathcal{R}}$ (where $\mathcal{R}$ is the root of $\text{FT}(st)$), $\text{QUERY}(t, \text{ROOT}(\text{FT}(st)))$ simply returns the length $|\mathbf{P}_{\mathcal{R}}|$. Hence, assume that $e = (u, v)$ lies on $\mathbf{P}_{\mathcal{R}}$ within the segment xy, where u is closer to s than v. Let $R = st \diamond e$. We first consider the simpler case where R also avoids the segment xy.

Let $\mathcal{X}$ be the segment child of $\mathcal{R}$ corresponding to the removal of the segment xy. The segment child $\mathcal{X}$ must exist as R itself is one candidate for $\mathbf{P}_{\mathcal{X}}$. Since $\mathbf{P}_{\mathcal{X}}$ avoids all edges of xy and $e \in xy$, it avoids e. Moreover, $\mathbf{P}_{\mathcal{X}}$ is a valid s to t path avoiding e, so $|\mathbf{P}_{\mathcal{X}}| \geq |R|$. Conversely, R itself is a candidate 1-decomposable path avoiding xy (since R avoids xy entirely) and $\mathbf{P}_{\mathcal{X}}$ is the shortest 1-decomposable path avoiding xy. So, $|\mathbf{P}_{\mathcal{X}}| \leq |R|$. Thus $|\mathbf{P}_{\mathcal{X}}| = |R|$. Consequently, in Line 10 we set $\textsc{Path} = |\mathbf{P}_{\mathcal{X}}|$.

We now analyze the case where R intersects xy. If R intersects xy both above and below e, we will give preference to its intersection below e in the analysis. The two possibilities are stated below:

4.5.1 R intersects xy above e at a vertex z

It means that R does not intersect xy below e. Assume that the detour of R starts at vertex z, and that $|R[z,t]| \in \left[(1+\varepsilon)^{i-1}, (1+\varepsilon)^i\right]$ for some $i \geq 1$. There exists a detour child $\mathcal{R}_i$ of $\mathcal{R}$ such that the detour of $\mathbf{P}_{\mathcal{R}_i}$ starts above e, say at the vertex w, and $\mathbf{P}_{\mathcal{R}_i}[w,t]$ has length at most $(1+\varepsilon)^i$. Since R itself is a candidate for $\mathbf{P}_{\mathcal{R}_i}$, such a path must exist. In our construction, we require the detour of $\mathbf{P}_{\mathcal{R}_i}$ to start as close to x as possible on the segment xy. Hence, w lies on or above z on $\mathbf{P}_{\mathcal{R}}$. In our algorithm, we set $\textsc{Path}$ to:

$$\begin{aligned}\textsc{Path} &= |\mathbf{P}_{\mathcal{R}_i}| \\ &= |\mathbf{P}_{\mathcal{R}_i}[s,w]| + |\mathbf{P}_{\mathcal{R}_i}[w,t]|\end{aligned}$$

But the detour of $\mathbf{P}_{\mathcal{R}_i}$ is of length at most $(1+\varepsilon)^i$. Thus, $|\mathbf{P}_{\mathcal{R}_i}[w,t]| \leq (1+\varepsilon)^i \leq (1+\varepsilon)|R[z,t]|$. Also, since the detour of $\mathbf{P}_{\mathcal{R}_i}$ starts at w on $\mathbf{P}_{\mathcal{R}}$, $\mathbf{P}_{\mathcal{R}_i}[s,w] = \mathbf{P}_{\mathcal{R}}[s,w]$. Substituting these two terms, we get:

$$\leq (1+\varepsilon)|R[z,t]| + |\mathbf{P}_{\mathcal{R}}[s,w]|$$

But $|\mathbf{P}_{\mathcal{R}}[s,w]| \leq |\mathbf{P}_{\mathcal{R}}[s,z]|$ as w lies at or above z on $\mathbf{P}_{\mathcal{R}}$. Thus, we get:

$$\leq (1+\varepsilon)|R[z,t]| + |\mathbf{P}_{\mathcal{R}}[s,z]|$$

But $\mathbf{P}_{\mathcal{R}}[s,z] = R[s,z]$ as the detour of R starts at z (See Part 3 of Figure 3). Thus, we get:

$$\begin{aligned}&\leq (1+\varepsilon)|R[z,t]| + |R[s,z]| \\ &\leq (1+\varepsilon)|R|\end{aligned}$$

4.5.2 R intersects xy below e, say at a vertex z

We first show that $|sv \diamond e| < |st \diamond e|$, implying $v \in L$. Since $st \diamond e$ passes through a vertex z on the segment xy, and z lies closer to v than to t, we expect $|sv \diamond e| < |st \diamond e|$. We now state the following lemma and prove this formally later in Appendix C.

▶ **Lemma 18.** *Let $e = (u,v)$ be the faulty edge closest to y on the segment xy. Let u be closer to x than y on the segment xy. $R = st \diamond e$ intersects xy below e, say at vertex z. Then $|sv \diamond e| \leq |st \diamond e|$. Similarly, if R intersects xy above e, say at w, and the subpath wu of xy does not contain any faults, then $|su \diamond F| \leq |st \diamond F|$.*

By the above lemma, v lies before t in L. In the worst case, v is the vertex preceding t in the sequence. So, using the induction hypothesis, $\textsc{Query}(v, \textsc{Root}(\textsc{Ft}(sv)))$ returns a $(1+(k-1)^3\varepsilon)$-approximation of $|sv \diamond e|$. We are now ready to bound the answer returned by our algorithm. In Line 12 we set

$$\textsc{Path} = |\mathbf{P}_{\mathcal{R}}[v,t]| + \textsc{Query}(v, \textsc{Root}(\textsc{Ft}(sv)))$$

Using Equation (1) and setting $Q = \mathbf{P}_{\mathcal{R}}[z,t]$, $S = xy$, we get:

$$= |Q| + |S[z,v]| + \textsc{Query}(v, \textsc{Root}(\textsc{Ft}(sv)))$$

Thus Path is of the form described in Lemma 16. We now describe the approximation ratio of each component of this path.

1. Q : After intersecting segment xy at z, the path R must continue along the shortest path $\mathbf{P}_{\mathcal{R}} = st$, so $|\mathbf{P}_{\mathcal{R}}[z,t]| = |R[z,t]|$. Thus, $Q = \mathbf{P}_{\mathcal{R}}[z,t]$ gives a 1-approximation of $|R[z,t]|$.
2. $S[z,v]$: Now, $|S[z,v]| \leq |S| = |xy|$. By Lemma 6, $|xy| \leq \varepsilon|\mathbf{P}_{\mathcal{R}}| \leq \varepsilon|R|$ (since $\mathbf{P}_{\mathcal{R}}$ is the shortest st path, $|\mathbf{P}_{\mathcal{R}}| \leq R$). Thus, $S = xy$ gives a $\overbrace{\varepsilon}^{\beta}$-approximation of $|R|$.
3. $\textsc{Query}(v, \textsc{Root}(\textsc{Ft}(sv)))$: Using Lemma 18, and the induction hypothesis, $\textsc{Query}(v, \textsc{Root}(\textsc{Ft}(sv)))$ returns a $\overbrace{(1 + (k-1)^3\varepsilon)}^{\alpha}$ approximation of $|sv \diamond e|$.

   Substituting $\alpha = 1 + (k-1)^3\varepsilon$, and $\beta = \varepsilon$ into Lemma 16, we get $\textsc{Path} \leq (1 + k^3\varepsilon)|R|$. This completes the correctness of our algorithm. We claim the following lemma:

▶ **Lemma 19.** *There is a $(1 + 27\varepsilon)$-approximate single fault tolerant distance oracle with $\tilde{O}(n\sqrt{n})$ space and $\tilde{O}(1)$ query time.*

One could perform a more detailed analysis to achieve a better approximation ratio than in the above lemma. However, we prioritize conciseness, since this approach extends naturally to the dual-fault scenario. We now consider the more general case of two faulty edges.

5 Extending our approach to 2 faults

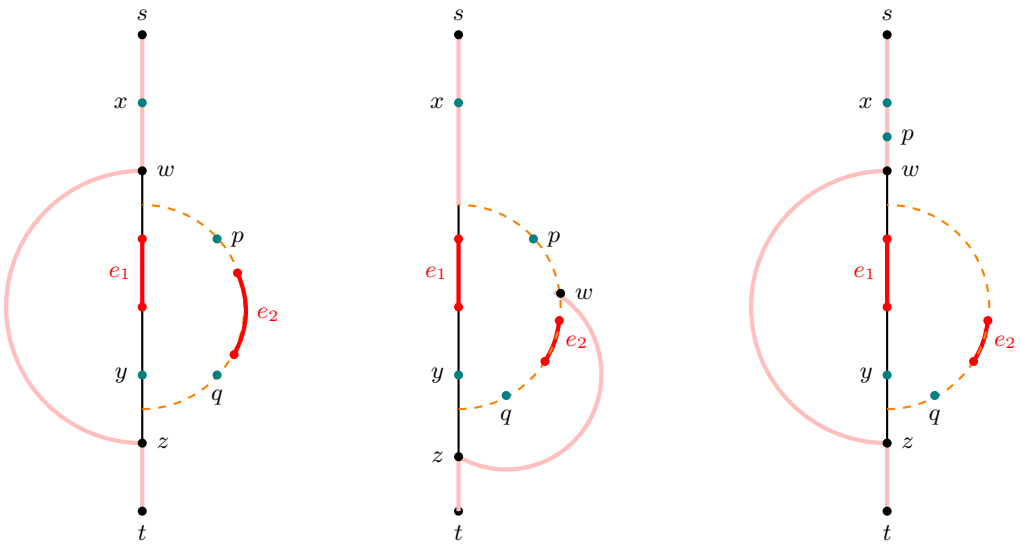
We extend our data structure and algorithm to handle dual faults by revisiting all components, including the Ft structure and the query algorithm. While the overall framework remains similar to the single-fault case, key differences arise when handling two faults. In the single-edge fault setting, a detour of $st \diamond e$ (see Definition 8) starts on st. However, for two edge faults, this definition is no longer sufficient. A path that avoids a faulty edge on the st path may itself contain the second faulty edge. Therefore, the notion of a detour must be refined. We extend the definition of detours (from Definition 8) to account for two edge faults.

▶ **Definition 20** (Detour of a path R avoiding two edges)**.** *Let $P_1 = st$ and let e_1 be the last faulty edge on P_1. Let P_2 be a path that avoids e_1 but contains the edge e_2. Let R avoid $\{e_1, e_2\}$. Let w be the last vertex of R on $\textsc{Prefix}(P_1, e_1) \cup \textsc{Prefix}(P_2, e_2)$. Then $R[w,t]$ is the detour of R and the detour starts at w.*

In the single fault case, we crucially used detours that start from a segment. If there is only one segment in question, then our Definition 9 holds. However, in a dual fault case, we may have to deal with two paths as alluded in the above definition. These two paths may have two different segments. Thus, the detour may start from the segment on the path P_1 (that contains the first fault e_1) or from the segment on the path P_2 (that contains the second fault e_2). Thus, we redefine the notion of a detour starting from a segment.

▶ **Definition 21** (Detour of a path R starting from a segment)**.** *Let $P_1 = st$ and let e_1 be the last faulty edge on P_1. Let P_2 be a path that avoids e_1 but contains the edge e_2. Let R avoid $\{e_1, e_2\}$. Let w be the last vertex of R on $\textsc{Prefix}(P_1, e_1) \cup \textsc{Prefix}(P_2, e_2)$. We say that the detour of R starts on the segment $seg(e_1, P_1)$ if*

1. *$w \in seg(e_1, P_1)$*
2. *$R[w,t]$ does not intersect with $seg(e_1, P_1) \cup seg(e_2, P_2)$ except at point w*



(a) Detour of R starts from segment xy.

(b) Detour of R starts from segment pq.

(c) w lies in both segments. However, as per our definition, the detour of R starts from the segment xy.

Figure 4 Illustration of different types of detours in path R under one or two edge failures. The straight line from top to bottom represents the st path (or path P_1), while the dashed path containing edge e_2 represents path P_2. e_1 lies on segment xy of P_1 and e_2 lies on segment pq of P_2. R is represented by the path coloured pink.

Note that the second condition means that the detour $R[w,t]$ does not intersect either $seg(e_1,P_1)$ or $seg(e_2,P_2)$, except at w.

Similarly, we say that R starts on $seg(e_2,P_2)$ if in the first condition above $w \in seg(e_2,P_2)$. If w lies both in $seg(e_1,P_1)$ and $seg(e_2,P_2)$, we give preference to the original shortest path P_1 and say that the detour starts on the segment $seg(e_1,P_1)$. See Figure 4a, Figure 4b, Figure 4c for illustrations.

We now describe the construction of $\textsc{Ft}(st)$ using the function MakeNode defined in Algorithm 3. This function is quite similar to Algorithm 1 – we have highlighted major changes in gray. This function takes one more extra parameter compared to Algorithm 1. We will describe its use later. Algorithm 1 constructs a tree that has two levels. But Algorithm 3 will construct a tree that has three levels – level 0, level 1 and level 2, where level 2 nodes are the leaves of the tree. We will now describe the construction in detail, starting with level 0 vertices. We make the root of $\textsc{Ft}(st)$ by invoking $\textsc{MakeNode}(G, st, \emptyset, 0)$. At the root $\mathcal{R}$, the primary path is $\mathbf{P}_{\mathcal{R}} = st$. At the root, we make a segment and detour children at level 1, which we describe next.

5.1 Level 1

The level 1 nodes constructed in our algorithm are nearly similar to Algorithm 1. The difference lies in the following step.

1. Segment Child : For a segment xy in $\mathbf{P}_{\mathcal{R}}$, we find the shortest **2-decomposable** path in the original graph G that survives in $G - xy$.

Algorithm 3 MakeNode$(H, P, B, depth)$.

Make a node $\mathcal{R}$ with $\mathbf{P}_{\mathcal{R}} = P$ and DetSegment$(\mathcal{R}) = B$
if *depth* < 2 **then**
  if *DetSegment*$(\mathcal{R})$ *is empty* **then**
    foreach *segment* xy *in* $\mathbf{P}_{\mathcal{R}}$ **do**
      Let $P(xy)$ be the shortest 2-decomposable path $Q_1 \odot e_1 \odot Q_2 \odot e_2 \odot Q_3$ in the original graph G, where each Q_j is a shortest path in G, such that $P(xy)$ survives in $H - xy$
      if $P(xy)$ *exists* **then**
        a child of $\mathcal{R} \leftarrow$ MakeNode$(H - xy, P(xy), \emptyset, depth + 1)$
      ▷ 1-decomposable detour children
      foreach $i = 0$ *to* $\log_{1+\varepsilon} nW$ **do**
        Let $\mathcal{Q} = \{P \diamond e_1 \mid e_1 \in xy\ \}$
        Let P_i be the path in $\mathcal{Q}$ whose detour starts closest to x on the segment xy and its detour joins below y on $\mathbf{P}_{\mathcal{R}}$ and the detour length $\leq (1+\varepsilon)^i$
        if P_i *exists (i.e.,* $\mathcal{Q} \neq \emptyset$*)* **then**
          a child of $\mathcal{R} \leftarrow$ MakeNode$(H, P_i, \{xy\}, depth + 1)$
      ▷ 2-decomposable detour children
      foreach $i = 0$ *to* $\log_{1+\varepsilon} nW$ **do**
        Let $\mathcal{L} = \{P \diamond \{e_1, e_2\} \mid e_1 \in xy$ *and* $e_2 \in (H - xy)\ \}$
        Let P_i be the path in $\mathcal{L}$ whose detour starts closest to x on the segment xy and its detour joins below y on $\mathbf{P}_{\mathcal{R}}$ and the detour length $\leq (1+\varepsilon)^i$
        if P_i *exists (i.e.,* $\mathcal{L} \neq \emptyset$*)* **then**
          a child of $\mathcal{R} \leftarrow$ MakeNode$(H, P_i, \{xy\}, depth + 1)$
  else
    Let DetSegment$(\mathcal{R}) = \{xy\}$
    foreach *segment* pq *in* $\mathbf{P}_{\mathcal{R}}$ **do**
      foreach $i = 0$ *to* $\log_{1+\varepsilon} nW$ **do**
        ▷ 2-decomposable detour children
        Let $\mathcal{Q} = \{P \diamond F \mid F = \{e, e'\}$ where $e \in xy$ and $e' \in pq\}$
        Let P_i be the path in $\mathcal{Q}$ whose detour starts closest to x on the segment xy and its detour joins below q on $\mathbf{P}_{\mathcal{R}}$ and the detour length $\leq (1+\varepsilon)^i$
        if P_i *exists* **then**
          a child of $\mathcal{R} \leftarrow$ MakeNode$(H, P_i, \{xy\}, depth + 1)$
        Let S_i be the path in $\mathcal{Q}$ whose detour starts closest to p on the segment pq and its detour joins below q on $\mathbf{P}_{\mathcal{R}}$ and the detour length $\leq (1+\varepsilon)^i$
        if S_i *exists* **then**
          a child of $\mathcal{R} \leftarrow$ MakeNode$(H, S_i, \{pq\}, depth + 1)$
return $\mathcal{R}$

2. Detour Child: Here we have two types of detour children.
   a. 1-decomposable detour child
      One is of the similar type as in Algorithm 1 where we have stored a detour path P_i whose detour starts closest to x on the segment xy and joins below the segment xy and its detour length is $\leq (1+\epsilon)^i$. Note that P_i, if it exists, is a 1-decomposable path. If such detour P_i exist, then we create a child $\mathcal{R}_i$ of $\mathcal{R}$ by calling $\textsc{MakeNode}(G, P_i, \{xy\}, 1)$, set its primary path to P_i. We call this detour a 1-decomposable detour as the path P_i is 1-decomposable. Note that in the single fault tolerant distance oracle, we only constructed 1-decomposable detour children and that is why we never gave them an explicit name. But for dual faults, we will construct 1-decomposable as well as 2-decomposable detour children (which we will describe in the next enumeration).
      Unlike segment children, $\mathcal{R}_i$ does not arise by removing a segment from $\mathbf{P}_{\mathcal{R}}$. Specifically, the detour of $\mathcal{R}_i$ starts from a segment of $\mathbf{P}_{\mathcal{R}}$. To mark this exception, we set $\textsc{DetSegment}(\mathcal{R}_i) = \{xy\}$, indicating that its primary path includes a detour originating in the segment xy.
   b. 2-decomposable detour child
      There will be another kind of detour child which is highlighted by gray in Algorithm 3. $\mathcal{L}$ contains a set of shortest paths avoiding two edges with one edge lying on the segment xy and the other edge can be anywhere in the graph. Let P_i be path in $\mathcal{L}$ whose detour starts as close to x on the segment xy, joins below y on $\mathbf{P}_{\mathcal{R}}$, with detour length at most $(1+\varepsilon)^i$. Note that P_i, if it exists, is 2-decomposable. We create a child $\mathcal{R}_i$ of $\mathcal{R}$ by calling $\textsc{MakeNode}(G, P_i, \{xy\}, 1)$, set its primary path to P_i, and define $\textsc{DetSegment}(\mathcal{R}_i) = \{xy\}$.

   As in the single fault case, there are $O(\log_{1+\varepsilon} nW)$ segment and $O(\log^2_{1+\varepsilon} nW)$ detour children at level 1.

5.2 Level 2

For a *segment* level-1 node, we recursively apply the same process used at the root. Since there are at most $O(\log_{1+\varepsilon} nW)$ segment nodes at level 1, each contributing up to $O(\log^2_{1+\varepsilon} nW)$ children at level 2. Therefore, the total number of level-2 nodes added while processing segment level-1 nodes is $O(\log^3_{1+\varepsilon} nW)$.

Consider a node $\mathcal{R}$ at level 1 with $\textsc{DetSegment}(\mathcal{R}) = \{xy\}$. By construction, $\mathbf{P}_{\mathcal{R}}$ is a path surviving the failure of an edge in xy. To handle a second fault, we iterate through every segment pq on the current path $\mathbf{P}_{\mathcal{R}}$. For each segment pq, we consider the set of paths $\mathcal{Q}$ that survive the failure of two edges $\{e_1, e_2\}$ (where $e_1 \in xy, e_2 \in pq$). From this set, we extract two different 2-decomposable detours for the next level. Note that these detours are defined with respect to the two segments xy and pq, and thus follow the detour starting rules outlined in Definition Definition 21

1. Let $P_i \in \mathcal{Q}$ be the path whose detour starts **closest to x** on the segment xy and its detour joins below q on $\mathbf{P}_{\mathcal{R}}$ and the detour length $\leq (1+\varepsilon)^i$. This path handles detour that start from the segment xy. We create a child $\mathcal{R}_i$ of $\mathcal{R}$ by invoking $\textsc{MakeNode}(G, P_i, \{xy\}, 2)$.
2. Let $S_i \in \mathcal{Q}$ be the path whose detour starts **closest to p** on the segment pq and its detour joins below q on $\mathbf{P}_{\mathcal{R}}$ and the detour length $\leq (1+\varepsilon)^i$. This path handles the detour that starts in the segment pq. We create a child $\mathcal{R}_i$ of $\mathcal{R}$ by invoking $\textsc{MakeNode}(G, S_i, \{pq\}, 2)$. Here, we update the set to $\textsc{DetSegment}(\mathcal{R}_i) = \{pq\}$ to explicitly record that the path depends on the detour originating from the segment pq.

For each segment $pq \in \mathbf{P}_{\mathcal{R}}$, we create at most $O(\log_{1+\varepsilon} nW)$ children at level 2 of $\text{FT}(st)$ (from both cases (1) and (2)). Since $\mathbf{P}_{\mathcal{R}}$ contains $\log_{1+\varepsilon} nW$ segments, each detour node $\mathcal{R}$ contributes $O(\log^2_{1+\varepsilon} nW)$ children. With $O(\log^2_{1+\varepsilon} nW)$ detour nodes coming from both 1-decomposable and 2-decomposable children at level 1, the total number of level-2 nodes arising from detour nodes is $O(\log^4_{1+\varepsilon} nW)$. We thus claim the following:

▶ **Lemma 22.** *There are $O(\log^4_{1+\varepsilon} nW)$ nodes in $\text{FT}(st)$.*

As in the single fault case, we claim that the path in each node of $\text{FT}(st)$ is 2-decomposable and we can represent each node of $\text{FT}(st)$ in $\tilde{O}(\sqrt{n})$ space. Thus, we claim the following lemma.

▶ **Lemma 23.** *For each t , $\text{FT}(st)$ can be represented using $\tilde{O}(\sqrt{n})$ space. Thus, the total space taken by $\text{FT}(st)$ over all $t \in V \setminus \{s\}$ is $\tilde{O}(n\sqrt{n})$. Moreover, the time taken to find if $e \in \mathbf{P}_{\mathcal{R}}$ is $\tilde{O}(1)$ and we can also report the segment of $\mathbf{P}_{\mathcal{R}}$ that contains e in $\tilde{O}(1)$ time.*

Due to space constraints, we refer the reader to the the full version of the paper [10] for query algorithm, its running-time analysis, and the proof of correctness for the two-edge fault case.

6 Conclusion and Open Problems

In this paper, we designed a $\text{SDO}(2)$ with $o(n^2)$ space and $\tilde{O}(1)$ query time. However, our algorithm does not extend to handle three or more edge faults. We explain the reason next.

Let $F = \{e_1, e_2, e_3\}$ be the set of faulty edges on the st shortest path. Suppose e_2 and e_3 lie on the segment xy of this path, and that $st \diamond F$ intersects st between e_2 and e_3 (see the pink path in the figure Figure 5). Furthermore, for every vertex z on the subpath v_1u_2 of st, $|sz \diamond F| \gg |st \diamond F|$. Thus, z cannot lie in any path that approximates $st \diamond F$ by a factor of $(1+\varepsilon)$.

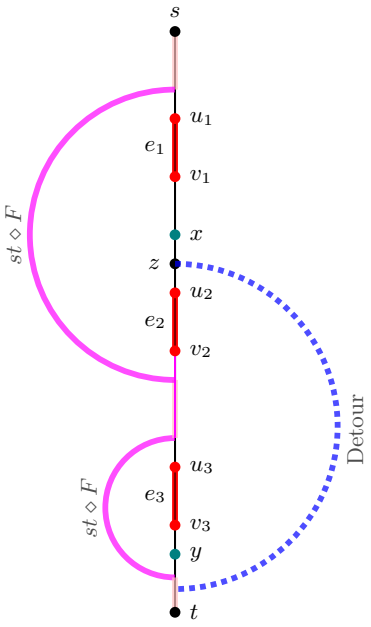


Figure 5 3 edge fault case.

To extend our dual fault strategy to handle three faults, we attempt to find a detour that starts as high as possible on xy and rejoins the st path below y. Suppose this detour begins above e_2 at a vertex z (see the blue path in the figure). If e_1 were absent, the path from z to s along st would suffice to reach s. However, the presence of e_1 introduces significant complications. By our assumption, $|sz \diamond F| \gg |st \diamond F|$, so this detour offers no advantage. In fact, in this configuration, it is crucial to find a detour that starts between e_2 and e_3 on the st path. However, identifying such a detour is far from straightforward.

We conclude with the following open questions. Can we design a data structure of size $o(n^2)$ that provides a good approximation for the shortest path under three faults? Can this approach be generalised to handle f faults?

References

1 Yehuda Afek, Anat Bremler-Barr, Haim Kaplan, Edith Cohen, and Michael Merritt. Restoration by path concatenation: fast recovery of MPLS paths. *Distributed Computing*, 15(4):273–283, 2002. doi:10.1007/S00446-002-0080-6.

2 Surender Baswana, Keerti Choudhary, Moazzam Hussain, and Liam Roditty. Approximate single-source fault tolerant shortest path. *ACM Transactions on Algorithms (TALG)*, 16(4):1–22, 2020. doi:10.1145/3397532.

3 Surender Baswana and Neelesh Khanna. Approximate shortest paths avoiding a failed vertex: Near optimal data structures for undirected unweighted graphs. *Algorithmica*, 66:18–50, 2013. doi:10.1007/S00453-012-9621-Y.

4 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, pages 88–94, 2000. doi:10.1007/10719839_9.

5 Aaron Bernstein and David Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 101–110, 2009. doi:10.1145/1536414.1536431.

6 Davide Bilò, Shiri Chechik, Keerti Choudhary, Sarel Cohen, Tobias Friedrich, Simon Krogmann, and Martin Schirneck. Approximate distance sensitivity oracles in subquadratic space. *TheoretiCS*, 3, 2024. doi:10.46298/THEORETICS.24.15.

7 Davide Bilò, Shiri Chechik, Keerti Choudhary, Sarel Cohen, Tobias Friedrich, and Martin Schirneck. Improved distance (sensitivity) oracles with subquadratic space. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 1550–1558. IEEE, 2024. doi:10.1109/FOCS61266.2024.00097.

8 Davide Bilò, Sarel Cohen, Tobias Friedrich, and Martin Schirneck. Near-optimal deterministic single-source distance sensitivity oracles. In *Accepted in Annual European Symposium on Algorithms, ESA 2021*, 2021.

9 Shiri Chechik, Sarel Cohen, Amos Fiat, and Haim Kaplan. $(1 + \epsilon)$-approximate f-sensitive distance oracles. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1479–1496, 2017.

10 Koustav Das and Manoj Gupta. Approximate single source dual fault tolerant distance oracle, 2026. arXiv:2607.02999.

11 Ran Duan, Jiayi Mao, Xiao Mao, Xinkai Shu, and Longhui Yin. Breaking the sorting barrier for directed single-source shortest paths. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pages 36–44, 2025. doi:10.1145/3717823.3718179.

12 Ran Duan, Jiayi Mao, Xinkai Shu, and Longhui Yin. A randomized algorithm for single-source shortest path on undirected real-weighted graphs. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 484–492. IEEE, 2023. doi:10.1109/FOCS57990.2023.00035.

13 Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 506–515, 2009. doi:10.1137/1.9781611973068.56.

14 Ran Duan and Hanlin Ren. Maintaining exact distances under multiple edge failures. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1093–1101. ACM, 2022. doi:10.1145/3519935.3520002.

15 Michael L Fredman and Dan E Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 719–725. IEEE, 1990. doi:10.1109/FSCS.1990.89594.

16 Michael L Fredman and Dan E Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

17 Manoj Gupta and Aditi Singh. Generic single edge fault tolerant exact distance oracle. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 72:1–72:15, 2018. doi:10.4230/LIPIcs.ICALP.2018.72.

18 Torben Hagerup. Improved shortest paths on the word ram. In *International Colloquium on Automata, Languages, and Programming*, pages 61–72. Springer, 2000. doi:10.1007/3-540-45022-X_7.

19 John Hershberger and Subhash Suri. Vickrey prices and shortest paths: What is an edge worth? In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 252–259, 2001. doi:10.1109/SFCS.2001.959899.

20 Merav Parter and David Peleg. Sparse fault-tolerant BFS trees. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 779–790, 2013. doi:10.1007/978-3-642-40450-4_66.

21 Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *European Symposium on Algorithms*, pages 121–137. Springer, 1996. doi:10.1007/3-540-61680-2_51.

22 Rajeev Raman. Recent results on the single-source shortest paths problem. *ACM SIGACT News*, 28(2):81–87, 1997. doi:10.1145/261342.261352.

23 Mikkel Thorup. On ram priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000. doi:10.1137/S0097539795288246.

A Proof of Lemma 6

Assume u is closer to s than v, i.e., $|P[s,u]| < |P[s,v]|$. Let i be the maximal index such that $|P[s,u]| \geq (1+\epsilon)^i$. There are two cases. If $(1+\epsilon)^{i+1} \leq |P[s,v]|$, then by definition, both u and v are net points of P, so $seg(e,P) = \{e\}$. Otherwise, $(1+\epsilon)^{i+1} > |P[s,v]|$. Then

$$|seg(e,P)| \leq (1+\epsilon)^{i+1} - (1+\epsilon)^i = \epsilon(1+\epsilon)^i \leq \epsilon|P[s,u]|.$$

By symmetry, either $seg(e,P) = \{e\}$ or $|seg(e,P)| \leq \epsilon|P[v,t]|$. If $|seg(e,P)| \leq \epsilon|P[v,t]|$, then $|seg(e,P)| \leq \epsilon|P[u,t]|$ since $|P[v,t]| < |P[u,t]|$. Therefore, $seg(e,P) = \{e\}$ or $|seg(e,P)| \leq \epsilon \min\{|P[s,u]|, |P[u,t]|\}$,which implies $|seg(e,P)| \leq \epsilon|P|$.

B Proof of Lemma 14

Consider an edge $e = (u,v)$ on a shortest path Q. Our objective is to identify the segment xy such that $e \in xy$. The endpoints x and y may either belong to the same decomposable subpath Q_i or to two different subpaths Q_i and Q_j, where $i \neq j$.

Depending on the length of the decomposable subpath Q_i, we distinguish the following cases:

B.1 Case 1

Both endpoints x and y lie on the same decomposed subpath Q_i whose length is at most $\sqrt{n}$. Since $|Q_i| \leq \sqrt{n}$, we store a dictionary at node $\mathcal{R}$ using $\tilde{O}(\sqrt{n})$ space, allowing constant-time membership queries for $e \in Q_i$. Hence, we can efficiently return the segment xy that contains the edge $e = (u, v)$.

B.2 Case 2

The segment endpoints x and y lie on the same decomposable path Q_i, but the path length satisfies $|Q_i| \geq \sqrt{n}$.

In this case, we represent Q_i implicitly as a concatenation of two shortest paths, xz and zy, where $z \in L$. We then examine the tree T_z, rooted at z, to determine whether the edge $e = (u, v)$ lies within either subpath. Let us assume that e is in T_z.

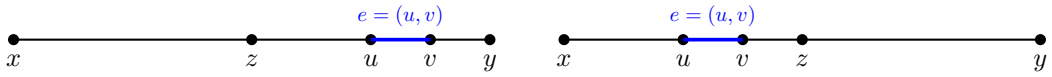


Figure 6 Illustration of subpath decomposition of P_i as xz and zy, where $z \in L$. Depending on the position of the edge $e = (u, v)$, it may lie in either subpath.

- If on T_z, $\text{LCA}(v, y) = v$ and $\text{LCA}(u, y) = u$, then $e = (u, v)$ lies in the subpath zy. See Figure 6
- If on T_z, $\text{LCA}(v, x) = v$ and $\text{LCA}(u, x) = u$, then $e = (u, v)$ lies in the subpath xz. See Figure 6

B.3 Case 3

Finally, consider the case where x lies on a decomposable path Q_i and y lies on another decomposable path Q_j, where $i \neq j$. We represent the path $\mathbf{P}_\mathcal{R}$ as

$\mathbf{P}_\mathcal{R} = Q_1 \odot e_1 \odot Q_2 \odot e_2 \odot Q_3$,

where each Q_i $(i \in \{1, 2, 3\})$ is a shortest path in G.

As illustrated in Figure 7, suppose the path P is decomposed as

$P = Q_i \odot e_1(a, b) \odot Q_k \odot e_2(c, d) \odot Q_j$.

Here, Q_i contains one of the segment endpoints x, and the path Q_i ends at vertex a. The edge $e = (u, v)$ lies in the middle decomposed path Q_k, which starts at b and ends at c. The other segment endpoint y lies in the decomposed path Q_j, which begins at d and ends at t.

If all the decomposed paths have lengths at most $\sqrt{n}$, we can determine whether x lies in Q_i, $e = (u, v)$ lies in Q_k, and y lies in Q_j using the method described in Case 1.(See B.1) However, if one or more of these paths have length greater than $\sqrt{n}$, we can identify a vertex $z \in L$ for which we have stored the corresponding tree T_z. Using T_z, we can perform the necessary LCA queries to determine the locations of x, y, and $e = (u, v)$.

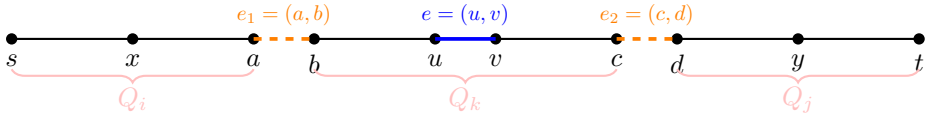


Figure 7 The segment endpoints x and y lie in different decomposed paths Q_i and Q_j, respectively.

Since there are $\log_{1+\epsilon} n$ segments, there can be at most $2 \log_{1+\epsilon} n$ segment endpoints. Hence, the total time required for this operation is $O(\log_{1+\epsilon} n)$.

C Proof of Lemma 18

Proof. By the triangle inequality,

$|sv \diamond e| \leq |sz \diamond e| + |zv \diamond e|$

Since z lies on the segment xy, and there is no faulty edge between z and v, we can write

$|zv \diamond e| = |zv|$

$\leq |sz \diamond e| + |zv|$

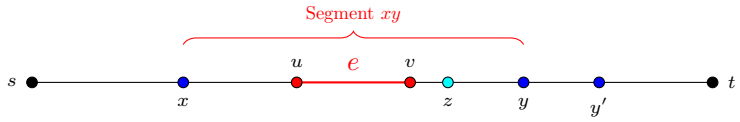


Figure 8 Proving $|zv| \leq |tz|$.

We now show that $|zv| \leq |tz|$ (See Figure 8). Suppose, for contradiction, that $|zv| > |tz|$. Let y' be the net-point closest to y originating from t in the path ty. Thus, $(1+\varepsilon)^{i+1} > |ty'| \geq (1+\varepsilon)^i$ for some i. Note that y may be equal to y'. Thus,

$|vt| = |tz| + |zv|$

$> 2|tz|$ (By assumption, $|zv| > |tz|$)

Since the path tz contains y', $|tz| > |ty'| \geq (1+\varepsilon)^i$. Thus, we get:

$|vt| \geq 2(1+\varepsilon)^i$

$\geq (1+\varepsilon)^{i+1}$

This implies that there must be a net-point originating from t in $y'v$ path. But there is no net point originating from t along the path $y'y$ (by construction) and between yv (since yv is a subpath of segment xy). Thus, we arrive at a contradiction. Hence, $|zv| \leq |tz|$. We now bound $sv \diamond e$.

$|sv \diamond e| \leq |sz \diamond e| + |zv|$

For our discussion above, $|zv| \leq |zt|$. Thus, we get:

$\leq |sz \diamond e| + |zt|$

$\leq |sz \diamond e| + |zt \diamond e|$

$= |st \diamond e| = |R|$ (Since R passes through z)

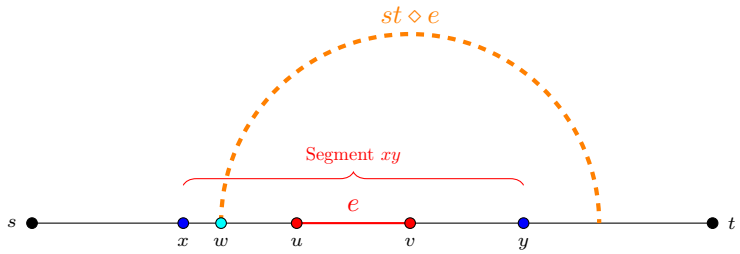


Figure 9 $|su \diamond e| \leq |st \diamond e|$.

For the second part of the lemma, using triangle inequality, we know that ,

$$|su \diamond F| \leq |sw \diamond F| + |wu \diamond F| \tag{2}$$
$$= |sw \diamond F| + |wu| \quad \text{(since } wu \text{ has no fault)} \tag{3}$$

Since R intersects xy above e at a vertex w

$$|st \diamond F| \leq |sw \diamond F| + |wt \diamond F| \tag{4}$$

As wu is a subpath of wt,

$$|wu| < |wt| \leq |wt \diamond F| \tag{5}$$

From (3) and (5), we get

$$|su \diamond F| < |sw \diamond F| + |wt \diamond F| \tag{6}$$
$$\leq |st \diamond F|. \tag{7}$$

◀