Near Optimal Dual Fault Tolerant Distance Oracle

Dipan Dey
IIT Gandhinagar, India

Manoj Gupta
IIT Gandhinagar, India

Abstract

We present a dual fault-tolerant distance oracle for undirected and unweighted graphs. Given a set F of two edges, as well as a source node s and a destination node t, our oracle returns the length of the shortest path from s to t that avoids F in $O(1)$ time with a high probability. The space complexity of our oracle is $\tilde{O}(n^2)$ [1], making it nearly optimal in terms of both space and query time.

Prior to our work, Pettie and Duan [SODA 2009] designed a dual fault-tolerant distance oracle that required $\tilde{O}(n^2)$ space and $O(\log n)$ query time. In addition to improving the query time, our oracle is much simpler than the previous approach.

2012 ACM Subject Classification Theory of computation; Theory of computation → Design and analysis of algorithms; Theory of computation → Shortest paths

Keywords and phrases Distance Sensitive Oracle, Dual Fault Distance Oracle

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.45

Related Version *Full Version*: https://arxiv.org/abs/2406.19709

1 Introduction

Graph theory is a valuable tool for modelling and solving many real-world problems. As real-world networks are susceptible to failures, such as faulty edges or vertices within the network, we must design a system that continues to operate in the presence of failures. In this paper, we focus on developing algorithms that effectively operate in the presence of faults. Consider the scenario where we aim to determine the shortest path from a source vertex to a destination vertex. However, we become aware that one of the edges in the graph is unavailable or faulty due to some reasons. Under these circumstances, our objective becomes finding an alternative shortest path that avoids the faulty edge.

Let us describe an abstract model in which we want to solve the above problem. We consider an undirected and unweighted graph, denoted as G. To facilitate efficient query processing, we preprocess this graph to construct a suitable data structure. This data structure is used to answer queries about shortest paths in G in the presence of faults. We assume that we will receive queries of the following form:

Query(s,t,F): Find length of the shortest path between vertices s and t in graph G, avoiding set of edges F.

The objective of our algorithm is to efficiently answer the above query by utilising the prepared data structures. After creating a data structure, while queried about Query(s,t,F), we run an algorithm (named query algorithm) to answer the query. This combination of data structure and query algorithm is commonly referred to as an *oracle*. Our paper focuses on calculating distances in the presence of faults. Consequently, we refer to our oracle as a *fault-tolerant distance oracle*.

[1] $\tilde{O}$ hides polylog n factor

© Dipan Dey and Manoj Gupta;
licensed under Creative Commons License CC-BY 4.0
32nd Annual European Symposium on Algorithms (ESA 2024).
Editors: Timothy Chan, Johannes Fischer, John Iacono, and Grzegorz Herman; Article No. 45; pp. 45:1–45:23

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We evaluate the fault-tolerant distance oracle based on two key parameters: (1) the space required by the data structure and (2) the query time, which refers to the time taken to respond to a query. To facilitate the description of our results in this context, we introduce a simple notation. We define a distance oracle that avoids a fault set of size at most f as $\textsc{Do}(f)$. When the source vertex s is fixed in the $\textsc{Query}$, the oracle is referred to as a Single Source Distance Oracle or $\textsc{Sdo}(f)$ for brevity.

In this paper, our focus is designing $\textsc{Do}(2)$ – i.e., designing fault-tolerant distance oracle that can handle fault set of size at most 2. We discuss the previous results obtained in this area in Figure 1.

Oracle	Space	Query time	Remarks	Ref
$\textsc{Do}(1)$	$\tilde{O}(n^2)$	$O(1)$	-	[7]
$\textsc{Do}(2)$	$\tilde{O}(n^2)$	$O(\log n)$	-	[9]
$\textsc{Do}(f)$	$\tilde{O}(n^{3-\alpha})$	$\tilde{O}(n^{2-(1-\alpha)/f})$	$\alpha \in [0,1]$ when the preprocessing time is $O(Mn^{3.376-\alpha})$ and edge weights are integral in the range $[-M,\dots,M]$.	[18]
$\textsc{Do}(f)$	$\tilde{O}(n^{2+\alpha})$	$O(n^{2-\alpha}f^2 + nf^{\omega})$	$\alpha \in [0,1]$ and ω is the matrix multiplication exponent [6, 16, 19, 11, 2]	[17]
$\textsc{Do}(f)$	$O(fn^4)$	$O(f^{O(f)})$	-	[10]
f	$O(n^2)$	$\tilde{O}(nf^{\omega-1})$	Edge weights are in the range $[1\dots W]$	[14]
f	$O(f^4 n^2 \log(nW))$	$O(c^{(f+1)^2} f^{8(f+1)^2} \log^{2(f+1)^2}(nW))$	Edge weights are in the range $[1\dots W]$	[8]

Figure 1 Relevant results for fault-tolerant distance oracles.

For $\textsc{Do}(2)$, the oracle of Pettie and Duan [9] is nearly optimal. It takes $\tilde{O}(n^2)$ space and $O(\log n)$ query time. However, as the authors of [9] also mentioned, their approach is overly complicated and requires extensive case analysis. Duan and Ren [10] proposed a fundamentally different approach. They designed an algorithm for $\textsc{Do}(f)$ that takes $O(n^4)$ space and $O(f^{O(f)})$ query time. For $\textsc{Do}(2)$, their query time is $O(1)$. Unlike [9], the main feature of their algorithm is that it involves a limited amount of case analysis. However, the space taken by their algorithm is $O(n^4)$. Recently, Dey and Gupta [8] designed an f-fault tolerant distance oracle with $\tilde{O}(n^2)$ space and $\tilde{O}(1)$ query time for fixed f. In this paper, using an approach similar to [8] and [10], we design a dual-fault tolerant oracle with $\tilde{O}(n^2)$ size and $O(1)$ query time. We use some randomisation techniques which helps us to reduce the query time. The main result of this paper is:

▶ **Theorem 1.** *For an undirected and unweighted graph, there is a dual fault-tolerant oracle that takes $\tilde{O}(n^2)$ space and answers each query in $O(1)$ time with high probability.*

Unfortunately, some tools we use in this paper do not works for weighted graphs.

2 Overview

In this section, we will at first give an overview of the method used by Duan and Ren in [10] to create an f-fault tolerant distance oracle with $O(n^4)$ size. After that, we will give an overview of our modifications to that approach to have an oracle for dual faults with $\tilde{O}(n^2)$ size and constant query time. For that, let us define some notations at first. The notation st represents the shortest path from vertex s to vertex t. The notation $|st|$ denotes the length of this path (same as the number of edges in the st path for unweighted graphs). The notation $st \diamond F$ denotes the shortest path from s to t while avoiding a set of edges $F = \{e_1, e_2\}$, and $|st \diamond F|$ corresponds to the length of that path.

2.1 Overview of the approach in [10]

Suppose we want to find the path $P = st \diamond F$. In [10], the authors introduced the concept of "maximiser", which will be discussed further in later sections. For now, we want to emphasize on the fact that the "maximisers" output either the length of the path P or a set of vertices serving as a "hitting set" of P. We will formally prove this later. Together with these "maximisers", they also used a structural theorem of shortest paths under edge failures to use the hitting set effectively. A path is said to be f-decomposable if it is the concatenation of at most $(f + 1)$ shortest paths in G. In [1], the authors show the following:

▶ **Theorem 2** (Theorem 1 in [1]). *For any set of failures F of size f, every shortest path from s to t avoiding F is f-decomposable.*

Let us now discuss an important aspect of this theorem. Let $P = st \diamond \{e_1, e_2\}$ be 2-decomposable and x be a vertex of the hitting set which lies on the path P. Since P is 2-decomposable, it contains at most three shortest paths, say A, B, and C, where P starts with the path A and ends with the path C. If x lies on B, then $P[s, x]$ and $P[x, t]$ are 1-decomposable. Thus, we have hit path P and reduced the problem into two smaller subproblems (in terms of decomposability). However, if x lies on A or C, then we have not reduced the problem. The whole game now is to design an algorithm that can find an $x \in B$. In [10], Ren and Duan design such an algorithm. Since our strategy is similar, let us define some notations that formalise our discussion.

▶ **Definition 3** (First and the last segment in a path and an intermediate vertex). *Let us assume that F is a set of size f and $P = st \diamond F$ is f-decomposable. Thus, P can be decomposed into at most $f + 1$ shortest paths. Let A and B be the first and the last shortest path from these $f + 1$ shortest paths. Thus, A is the shortest path that starts from s, and B is the shortest path that ends at t. A is called the first segment of the path P and B is called the last segment of the path P. If a vertex $x \in P$ does not lie in the first and last segments, then it is called an intermediate vertex.*

Before we describe further, let us at first define some notations.

▶ **Definition 4.** *A path sx is said to be* **intact from failures** $\{e_1, e_2\}$ *if the endpoints of e_1 and e_2 do not lie strictly between*[2] *s and x on the path sx. Similarly, if the subtree under a node x in T_s, that is $T_s(x)$, do not contain any endpoints of e_1 and e_2, it is intact from failures $\{e_1, e_2\}$.*

The above definition may appear natural. However, it is not. Usually, when we say that a path or tree is intact from failures $\{e_1, e_2\}$, we mean that it does not contain e_1 and e_2. In contrast, the above definition does not allow the endpoints of e_1 and e_2. Though never emphasised, this fact is used crucially in the algorithm of Ren and Duan [10]. We now define the notion of a *clean* vertex introduced in [10].

▶ **Definition 5.** *A vertex x is called s-clean from $\{e_1, e_2\}$ if sx and $T_s(x)$ are intact from failures $\{e_1, e_2\}$.*

Duan and Ren used the above "clean" vertices crucially in [10]. Let $P = st \diamond F$ be f-decomposable. As mentioned above, we want to find an intermediate vertex x on P, after which we can recurse on $P[s, x]$ and $P[x, t]$. To this end, [10] designed an algorithm

[2] [10] did not require the endpoints to be strictly inside. However, we require it to avoid some corner cases.

QUERY, which is initially called using the parameters (s, t, F, f). The main component of this algorithm is the function HITSET, a function that uses maximisers. HITSET outputs $O(\text{poly}(f))$ edges H and an upper bound on $|P|$, which is stored in L. The function HITSET has the following properties: (1) Either $L = |P|$ or (2) One of the endpoints of the edges in H hits P and is an intermediate vertex. Thus, we either find our answer or recurse on two parts in Line 6 of Algorithm 1. One can check that if the running time of HITSET is polynomial in f, then the running time of the query algorithm is $O(f^{O(f)})$. In our case, this is $O(1)$ since $f = 2$.

Algorithm 1 Query Algorithm for the Exact Distance Oracle: QUERY(s, t, F, f).

1 **if** $st \cap \{e_1, e_2\} = \emptyset$ **return** $|st|$
2 **if** $f = 0$ **return** ∞
3 $(L, H) \leftarrow$ HITSET(s, t, F);
4 ANS $\leftarrow L$;
5 **for** *each x in H* **do**
6 ANS $\leftarrow \min\{\text{ans}, \text{QUERY}(s, x, F, f-1) + \text{QUERY}(x, t, F, f-1)\}$;
7 **return** ANS;

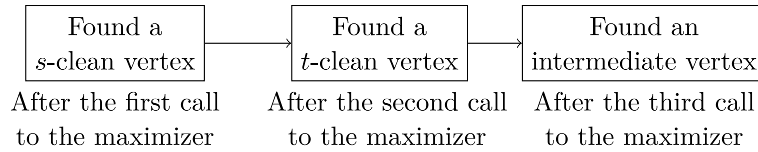

Figure 2 After at most three calls to the maximizer, we are sure to find an intermediate vertex.

We now give an overview of how to find an intermediate vertex. Let us assume the worst case when the output of our maximisers always hit the path P. The key to finding an intermediate vertex is first finding a clean vertex from the source and the destination, i.e., an s-clean and t-clean vertex. The authors first show how to design an appropriate maximiser that ensures we obtain a clean vertex. Once a s-clean and t-clean vertex are found, authors design an appropriate maximiser that gives us an intermediate vertex. This approach is succinctly described in the Figure 2. This figure illustrates behaviour of the HITSET algorithm in the worst case. In the best case, the first call to the maximiser may give us an intermediate vertex.

Though we have hidden many technical details, we want the reader to note the steps required to find an intermediate vertex in the worst case. Also, to use their maximisers, they needed $O(n^4)$ space which we reduced in this paper for 2 faults.

2.2 Overview of our approach

We define some new notations which will help us in describing our approach. Suppose we want to find $st \diamond F$ where $F = \{e_1, e_2\}$.

▶ **Definition 6** (Primary and Secondary path). *The shortest st path is referred to as the primary path, which includes the edge e_1. The secondary path is $st \diamond e_1$.*

Following the above definition, we assume that the primary path contains e_1 (without loss of any generality). If the primary path contains none of the edges e_1 and e_2, then finding $st \diamond \{e_1, e_2\}$ is trivial. If the primary and the secondary path do not contain e_2, then we can

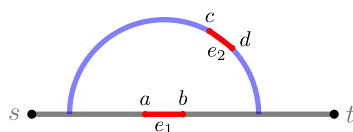

Figure 3 The primary path contains e_1 and the secondary path contains e_2.

(a) Detour starts at x on primary path.

(b) Detour starts at x on secondary path.

Figure 4 Start of the detour.

use the single edge fault-tolerant data structure of Demestrescu et al. [7] to find $|st \diamond e_1|$. This is also an easy case. For the rest of the paper, we will assume that we are in one of the two cases:

1. e_1 and e_2 both lie on the primary path
2. e_1 lies on the primary path and e_2 lies on the secondary path.

To apply our algorithm, we should quickly identify in which case we are operating. For this, we will introduce small weights to our originally unweighted graph similar to [4, 13, 15, 12]. This will ensure unique shortest paths between any two vertices in the graph, even after the occurrence of two faults. Let us abuse notation and use $|st|$ as the weight of the shortest path in this weighted graph. Also, let us define $w(e)$ as the weight of edge e in this new graph. To find if e_1 lies on the st path, we just need to check if $|st| = |se_1| + w(e_1) + |e_1t|$. Note that we can fetch these quantities in $O(1)$ time if we have stored the shortest path tree from each vertex in the graph. So, we can check if both e_1 and e_2 lie on the st path in $O(1)$ time. Similarly, we can find if e_2 is in secondary path by checking is $|st \diamond e_1| = |se_2 \diamond e_1| + w(e_2) + |e_2t \diamond e_1|$. Again, all the quantities can be found in $O(1)$ time using the one fault data structure of [7]. Henceforth, we will assume that we know the case in which we are operating.

We will show later that case (1) (both faults lie on the primary path) is similar to one of the subcases of the case (2) (e_1 on the primary path and e_2 on the secondary path). The bulk of our paper will be devoted to case (2). Henceforth, we will always assume the following setting: we want to find the shortest path from s to t avoiding $e_1 = (a, b)$ and $e_2 = (c, d)$. We will assume that $e_1 \in st$ and $e_2 \in st \diamond e_1$ where a and c are near to s on st and $st \diamond e_1$ paths respectively (See Figure 3). Let $P = st \diamond \{e_1, e_2\}$ be a 2-decomposable path. We now define the notion of the detour of the path P.

▶ **Definition 7** (Detour of a path P, start and end of a detour). *The detour of the path P is $P \setminus (st \cup st \diamond e_1)$. The start of the detour is the first vertex on P, after which P deviates from $st \cup st \diamond e_1$. If this vertex is on st, then we say that the detour starts on the primary path. If it is on $st \diamond e_1 \setminus st$, then we say that it starts on the secondary path. Similarly, we can define the end of the detour.*

Note that the above definition of detour differs slightly from what is usually used in literature. Specifically, the start of the detour is generally defined as the vertex on the path P, after which P deviates from st. In our definition, the detour can start on the primary path or the secondary path. Similarly, the detour may end on the primary or the secondary path.

In this paper, we use an approach similar to that used in [7]. Essentially, our objective is to identify vertices along the path $P = st \diamond \{e_1, e_2\}$ that are "close to the detour" from the vertices s and t. Before describing what we exactly mean by "close to the detour", let us define a suitable set of vertices called landmark vertices. These landmark vertices play a crucial role in the forthcoming definitions.

▶ **Definition 8** (Landmark vertices). *Let $\mathcal{L}_i$ be the set of vertices sampled from V with a probability of $\frac{c \log n}{2^i}$, where $0 \leq i \leq \log n$ and $c > 1$ is a constant. The size of $\mathcal{L}_i$ is $\tilde{O}\left(\frac{n}{2^i}\right)$ with a probability $\geq (1 - \frac{1}{n^c})$.*

We are now ready to describe our main contribution in this paper via the following two definitions:

▶ **Definition 9** (Close vertex to a faulty edge from s). *Let $P = st \diamond \{e_1, e_2\}$. A vertex $x \in P$ is called close vertex to edge e_1 from s if $x \in \mathcal{L}_\ell$ (for some $\ell \in [0, \log n]$), sx avoids $\{e_1, e_2\}$ and $|xe_1 \diamond \{e_1, e_2\}| = O(2^\ell)$. The definition is similar for close vertex to e_2. Similarly, we can define close vertex to a faulty edge from t.*

In most parts of the paper, we will find a close vertex from s. So, for brevity, we will drop the term "from s" where it is clear from the context that we are finding a close vertex from s. We aim to find a close vertex with one more special property. Next, we define the detour close vertex or D-close vertex. This will be close to faulty edge e_1 or e_2 depending on where the detour starts.

▶ **Definition 10** (A detour close or D-close vertex to a faulty edge from s). *Let $P = st \diamond \{e_1, e_2\}$. Let us assume that the detour of P starts on the primary path. A vertex $x \in P$ is called D-close to e_1 if x is close to e_1 and the detour of P starts in $xe_1 \diamond \{e_1, e_2\}$ path ($xe_1 \diamond \{e_1, e_2\}$ represents the path from x to e_1 on the primary path). If the detour of P starts on the secondary path, the definition of D-close to e_2 is similar (with $xe_1 \diamond \{e_1, e_2\}$ replace by $xe_2 \diamond \{e_1, e_2\}$, which represents the path from x to e_2 on the secondary path). Similarly, we can define a D-close vertex from t.*

Our central intuition is that once we have found a D-close vertex from s and t, we can use Ren and Duan's approach to find an s-clean or t-clean vertex. Note that finding a D-close vertex is not the main aim. The aim is to find an intermediate vertex. If we find a clean or intermediate vertex at any point in our algorithm, we abandon our search for a D-close vertex as our job is already done. We now design our version of HitSet algorithm in the Algorithm 1. In Figure 5, we show behaviour of our HitSet algorithm in the worst case: the first call may give us a D-close vertex from s. The second call gives us a D-close vertex from t. After that, we will find either an s-clean vertex or a t-clean vertex, and the remaining figure is same as Figure 5.

In essence, our result implies we should first find a vertex on P that is D-close to the fault. Similar kind of strategy can be found in [7]. This strategy not only decreases the query time but also improves the space taken by the algorithm. Finding a D-close vertex is easy when there is one fault. In our paper, we show how to find a D-close vertex (using maximisers) when there are two faults. Unfortunately, we will not be able to use the maximisers of Ren

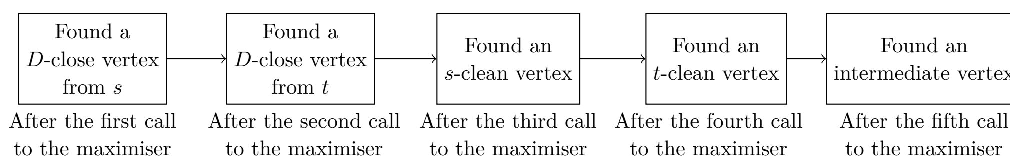

Figure 5 After at most five consecutive calls to the maximiser, we are sure to find an intermediate vertex.

and Duan[10] as its size is $O(n^4)$. We will design new maximisers different from [10], taking just $\tilde{O}(n^2)$ space. In fact, all the maximisers used in our HitSet algorithm take $\tilde{O}(n^2)$ space. This completes the description of our overview.

3 Preliminaries

Till now, we have discussed about the shortest path and shortest distance between any two vertices. In our paper, we also use notion of the shortest distance between a vertex and an edge. For an edge $e = (u, v)$ and a vertex s, $|se|$ is defined as the minimum of $|su|$ and $|sv|$. Also, $|se \diamond F| = \min\{|su \diamond F|, |sv \diamond F|\}$. When considering an arbitrary path P, in which vertex u appears before vertex v, $P[u, v]$ refers to the subpath from u to v, and $|P[u, v]|$ represents the length of this subpath. Also, $V(e)$ denotes the set containing both the endpoints of e, i.e., $V(e) = \{u, v\}$. T_s denotes the shortest path tree rooted at s. For any vertex x in T_s, $T_s(x)$ represents the subtree rooted at node x in T_s. In our algorithm, we may need to quickly find the least common ancestor (LCA) of any pair of vertices u and v in T_s. We will utilise the following result to find the LCA efficiently:

▶ **Lemma 11** (See [3] 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 queries in $O(1)$ time.*

We will use the following lemma to show the existence of landmark vertex (defined in Definition 8) with high probability on sufficiently long paths. We state this lemma without proof.

▶ **Lemma 12.** *Let U_i be the set of paths such that length of each path is $\Omega(2^i)$. If size of U_i is poly(n), then with a probability $\geq 1 - 1/n$, for all $0 \leq i \leq \log n$ and for each path $P \in U_i$, there exists a vertex in $\mathcal{L}_i$ that hits P.*

We use another concept from the works of [5] which is named as *trapezoid* of a path or the near vertices of a path. Before that, let us describe another notion below:

▶ **Definition 13** (Ball from a vertex of length ℓ, $\text{Ball}_G(u, \ell)$). *A ball from a vertex u of length ℓ is a set containing all the vertices at distance $\leq \ell$ from u. We denote this ball by $\text{Ball}_G(u, \ell)$. Formally, $\text{Ball}_G(u, \ell) = \{v \mid |uv| \leq \ell\}$.*

Let $P = st \diamond F$. Let us now define trapezoid of that path.

▶ **Definition 14** (Trapezoid of P in $G \setminus F$, $\text{Trapezoid}_{G\setminus F}(P)$). *Let P be an arbitrary path from s to t in $G \setminus F$. Define the trapezoid of P in $G \setminus F$ as:*

$$\text{Trapezoid}_{G\setminus F}(P) = \Big(\bigcup_{u \in P\setminus\{s,t\}} \text{Ball}_{G/F}(u, \epsilon \min\{|P[s, u]|, |P[u, t]|\}) \Big) \setminus \{s, t\}$$

where $\epsilon \geq 0$ is a parameter. In words, for each vertex $u \in P$, we include all vertices that are in the ball of length $\epsilon \ell_u$ in $G \setminus F$ where ℓ_u is the minimum of the length of u from s and t on path P. All these vertices lie in the trapezoid of P. A vertex is far away from P if it does not lie in the trapezoid of P.

Chechik et al.[5] introduced the notion of the trapezoid and designed a data structure FT. Though we will not use the FT data structure in our paper, we will crucially use the notion of a trapezoid. In [10], the authors introduced a technique called the maximiser for a pair of vertices s and t.

$$\mathbb{D}(s,t,x,y,b_1,b_2) = \text{argmax}_{F \in E^2} \{\text{conditions depending on } s,\ t,\ x,\ y,\ b_1 \text{ and } b_2\ \} \tag{1}$$

In the above maximiser, s is the source, and t is the destination. Variables x and y can be any two arbitrary vertices. The boolean variables b_1, b_2 gave some extra conditions on the pair of faults they are considering. Let e_1^* and e_2^* be the edges returned (or stored) by the above maximiser. Note that we can also store the path length $|st \diamond \{e_1^*, e_2^*\}|$ without increasing the size.

We first describe the power of these maximisers. Let us assume that we want to find the shortest path from s to t avoiding $\{e_1, e_2\}$. If $\{e_1, e_2\}$ satisfy the conditions of the maximiser $\mathbb{D}$, then we can show that edges returned by the maximiser $\mathbb{D}$ hit the path $st \diamond \{e_1, e_2\}$ or we have got the length of the path $st \diamond \{e_1, e_2\}$.

▶ **Lemma 15.** *Let $P = st \diamond \{e_1, e_2\}$ and $\mathbb{D}(s,t,x,y) = \{e_1^*, e_2^*\}$. If $\{e_1, e_2\}$ satisfies conditions of $\mathbb{D}$, then either e_1^* or e_2^* lie on P or $|P| = |st \diamond \{e_1^*, e_2^*\}|$.*

Proof. Let us assume that neither e_1^* nor e_2^* lies on P. Since $\{e_1, e_2\}$ also satisfies the condition given in $\mathbb{D}$, $|P| \leq |st \diamond \{e_1^*, e_2^*\}|$. Since P also avoids e_1^* and e_2^*, $|st \diamond \{e_1^*, e_2^*\}| \leq |P|$. This implies $|P| = |st \diamond \{e_1^*, e_2^*\}|$. ◀

Using the above lemma, either we find the length of the path $st \diamond \{e_1, e_2\}$ or we get a very small set that hits P. We now formally describe the maximisers created in [10] and how they use the notions of s-clean and t-clean vertices. Let *bits* be an array of size 2 (we replace b_1, b_2 used by them by *bits* for our presentation). Each cell in bits can either be 0 or 1. We will now define four maximisers $\mathbb{D}_{bits}(s,t,x,y)$ depending on the value in *bits*. The conditions of the maximisers are as follows:(1) sx and ty are intact from faults and (2) If $bits[0] = 1$, then x is s-clean. Similarly, if $bits[1] = 1$, then y is t-clean. We give the complete definition of one of the maximisers, $\mathbb{D}_{01}(s,t,x,y) = \text{argmax}_{\{e_1^*, e_2^*\} \in E^2} \{sx \text{ is intact from faults } \{e_1^*, e_2^*\} \text{ and } y \text{ is } t\text{-clean from } \{e_1^*, e_2^*\}\ \}$.

In the above definition, $s, t, x, y \in V$. Thus, the total size of $\mathbb{D}_{01}$ is $O(n^4)$. It is easy to observe that all the other three maximisers will also be of size $O(n^4)$. We have already discussed in Section 2 about how these maximisers are used. In this paper, we have modified these maximisers to have an oracle with size $\tilde{O}(n^2)$.

4 Oracle for two faults

Let us reiterate our setting: we want to find the shortest path from s to t avoiding $e_1 = (a, b)$ and $e_2 = (c, d)$. We will assume that $e_1 \in st$ and $e_2 \in st \diamond e_1$ where a and c are near s on st and $st \diamond e_1$ paths, respectively. Let $P = st \diamond \{e_1, e_2\}$ be a 2-decomposable path.

First, we describe some basic data structures used in our algorithm. As mentioned in Section 3, we will add small weights to each edge to ensure all the shortest paths are unique. Let G' be the graph obtained after adding weights. In G', we will find the shortest path tree from each vertex. The space taken by all shortest-path trees is $O(n^2)$. Similarly, whenever we need to find $st \diamond e_1$, we will apply the algorithm of [7] on G'. Again the algorithm in [7] takes $O(n^2)$ space. Apart from this data structure, we will never use G' in any of our other data structures. For the rest of the paper, our graph is unweighted.

Our new data structure is the maximiser $\mathbb{D}_{bits}(s, t, dist, clean)$. It stores the pair of edges (e_1^*, e_2^*) that maximises the distance from s to t subject to conditions determined by various parameters. We will also store the length of $st \diamond \{e_1^*, e_2^*\}$. Formally,

$$\mathbb{D}_{bits}(s, t, dist, clean) = \arg\max_{F \in E^2} \{\text{conditions that depend on } bits, s, t, dist, \text{ and } clean\} \quad (2)$$

One of the common conditions in all our maximisers in this section is as follows:

Common Condition:
A pair of edge (e, e') will be considered in maximiser only if $e \in st$, $e' \in st \diamond e$. (3)

Note that this condition is satisfied by the pair (e_1, e_2). We now describe other parameters of the maximiser. *bits*, *dist*, and *clean* are sets of size two implemented using arrays. Each element of *bits* can either be 0,1, or 2. If $bits[0] = 0$, then we have not found a D-close or s-clean vertex from s. If $bits[0] = 1$, then we have found a D-close vertex from s. Moreover, if $bits[0] = 2$, then we have found an s-clean vertex. We store the corresponding s-clean vertex in $clean[0]$. The same conditions hold for $bits[1]$ and $clean[1]$, which tells us whether we have found a D-close vertex from t or a t-clean vertex and stores the t-clean vertex if found. $dist[0]$ and $dist[1]$ normally store the distances from s and t respectively. There are nine versions of our maximiser $\mathbb{D}$, which depend on the value of *bits*. We will define them on the fly, as and when our algorithm needs them. We will discuss the space taken by these maximisers and our oracle in Appendix A. Before we describe our algorithm, let us take care of some corner cases.

We will always use maximiser so that $\{e_1, e_2\}$ satisfies its conditions. Let $\{e_1^*, e_2^*\}$ be the edges returned by the maximiser. If both e_1^* and e_2^* do not lie on P, then using Lemma 15, $|P| = |st \diamond \{e_1^*, e_2^*\}|$. Thus, we can easily find $|P|$ in this case. In the worst case, e_1^* and/or e_2^* may hit P. We will describe our algorithm for the worst case henceforth.

In our algorithm, let us assume that we have found a vertex x which is one of the endpoints of e_1^* or e_2^* and also lies on the path P. First, we check if x lies in the first or last segment of P or x is an intermediate vertex. For this, we need to check if sx (and tx) is intact from faults $\{e_1, e_2\}$. Using Lemma 11, this can be done in $O(1)$ time using appropriate LCA query in T_s (and T_t). Similarly, we can check if $T_s(x)$ contain faults $\{e_1, e_2\}$ in $O(1)$ time. Thus, processing x takes $O(1)$ time.

If x happens to be s-clean, we will set $clean[0] = x$ in the subsequent calls to the maximiser. If x is D-close from s, then we can recurse our algorithm on the path $P[x, t]$ on which x is acting as the source. Formally, if x is D-close from s, our answer is $|sx| + \text{QUERY}(x, t, \{e_1, e_2\}, 2)$. It may be that the primary path xt contains both the faults e_1 and e_2. In that case, we will use our algorithm in Appendix C to find $|P[x, t]|$. Similarly, if the secondary path from x to t does not contain any faulty edge, then we can find $|xt \diamond \{e_1, e_2\}|$ using the single fault algorithm of [7]. Again, these are easy cases for us. Henceforth, we assume that for any D-close vertex x, the primary and the secondary path from x to t contain faulty edges.

Finally, we will describe our approach to finding a D-close vertex from s, an s-clean vertex, and the intermediate vertex. Approach to finding these vertices from the destination side t is symmetrical.

5 Finding a D-close vertex from s

By definition, a D-close vertex can be on the primary path or the secondary path depending on where the detour of P starts. We deal these cases seperately.

5.1 The detour starts on the primary path

We first describe the maximiser. Since we have still not found a D-close vertex from s, $bits[0] = 0$. Our maximiser is $\mathbb{D}_{0\alpha}(s, t, dist, clean)$, where α can be $\{0, 1, 2\}$. We set $dist[0] = O(2^{\lfloor \log |se_1| \rfloor})$.

The conditions of the maximiser are as follows:

$$\mathbb{D}_{0\alpha}(s, t, dist, clean) = \arg\max_{F \in E^2} \Big\{ dist[0] \text{ distance from } s \text{ on the primary path is intact from } F \text{ and other conditions based on } \alpha \text{ from } t \text{ side.} \Big\} \tag{4}$$

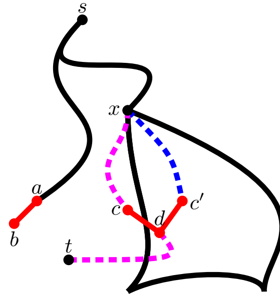

Figure 6 If $d \in T_s(x)$, the paths $sd \diamond e_1$ is not a prefix of $st \diamond e_1$.

Let x be an endpoint of an edge returned by the maximiser. If x is an intermediate vertex, we abandon our search for D-close vertex. So, let us assume that x lies on the first segment of P. We will show that x lies on the st path or x is s-clean.

▶ **Lemma 16.** *Let x be an endpoint of an edge returned by the maximiser (4) such that sx is intact from failures $\{e_1, e_2\}$. If x does not lie on the st path, then x is an s-clean vertex.*

Proof. Let us assume that $x \notin st$. We will now show that x is s-clean. Since sx is intact from failures $\{e_1, e_2\}$, we only need to show that endpoints of e_1 and e_2, i.e., a, b, c, d do not lie in $T_s(x)$.

Since x is not in the st path and hence not in the sa path, the two subtrees $T_s(x)$ and $T_s(a)$ are disjoint. Hence, $a, b \notin T_s(x)$.

Now, if $c \in T_s(x)$, then the path sc is intact from failures and is a subpath of the secondary path (as we assumed that the secondary path passes through e_2 and hence c). Also, x is in the path P. So, in this case, the detour will start in the subpath xe_2 on the secondary path. This is a contradiction.

If $d \in T_s(x)$, then since $c \notin T_s(x)$, there exists some c' such that $c'd$ is an edge of $T_s(x)$. Consider the path from s to x concatenated with the path from x to d in $T_s(x)$ (See Figure 6). This concatenated path represents the shortest path from s to d and avoids e_1. Additionally, it should be noted that T_s was constructed by adding small weights to each edge. Therefore, $|sd \diamond e_1| = |sd|$ is unique and does not pass through e_2. Furthermore, it should be remembered that we found $st \diamond e_1$ in the same weighted graph, and thus $st \diamond e_1$ is also unique and passes through e_2 (we assumed at the start of Section 4 that $e_2 \in st \diamond e_1$). However, according to the property of shortest paths, $sd \diamond e_1$ should be a subpath of $st \diamond e_1$. This leads to a contradiction, implying that $d \notin T_s(x)$. ◀

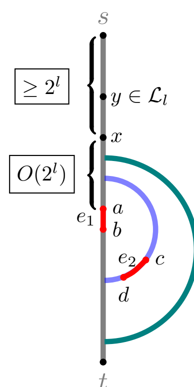

Figure 7 Finding a D-close vertex.

If x is not an s-clean vertex, then using Lemma 16, x lies on st path. We will now show how to find a D-close vertex from s using x. Let us assume that $\ell = \lfloor \log |se_1| \rfloor$. By the first condition of the maximiser (4), $|sx| \geq 2^\ell$. Also, since $|se_1| = O(2^\ell)$, $|xe_1| = O(2^\ell)$. We now find the first vertex, say y, from x on the xs path that lies in $\mathcal{L}_\ell$. Using Lemma 12, such a vertex exists with a high probability within a distance 2^ℓ from x. By construction, the distance $|ye_1|$ on the primary path is also $O(2^\ell)$. This implies that y is *close* to e_1. We now show that y is D-close to e_1 from s.

To this end, remember that sx is intact from failures and x lies on P as well as st path. Thus, by construction, even sy is intact from failures and y lies in P as well as st path. This implies that the detour of P starts on the primary path st after the vertex y or on the subpath ye_1 of st. Thus, y is D-close to e_1 from s.

5.2 The detour starts on the secondary path

We aim to find a D-close vertex to the edge e_2. The reader will see that this is a challenging case. In this case, we will use a different maximiser. We set $dist[0] = O(2^{\lfloor \log \min\{|se_1|, |se_2 \diamond e_1|\} \rfloor})$. Note that $se_2 \diamond e_1$ will not pass through e_2. Thus, the single fault data structure of [7] can find it in $O(1)$ time. The conditions in the maximiser are as follows:

$$\mathbb{D}_{0\alpha}(s, t, dist, clean) = \arg\max_{F \in E^2} \{ dist[0] \text{ distance from } s \text{ on the primary path and } \textbf{secondary} \text{ path is intact from } F \text{, other conditions based on } \alpha \text{ from } t \text{ side.} \} \quad (5)$$

Some remarks are in order. Specifically, we want the reader to note the use of primary and secondary paths in the maximiser. Let (e, e') be the pair of edges that satisfy the common condition of Equation (3). For this pair, the primary path is st (it always remains the same), and the secondary path is $st \diamond e$ (the secondary path changes for each pair). We demand that $dist[0]$ from both the primary and the secondary paths are intact from failures. The reader can check that the pair e_1, e_2 satisfies the conditions of the maximiser.

Let x be the endpoint of an edge returned by the maximiser such that it lies in the first segment of P. Let us assume that x is not s-clean. We will now show how to find a D-close vertex using x. There are two cases here. Let us look at both of them, starting with the easier case.

(1) $|se_2 \diamond e_1| \leq |se_1|$: This case is similar to the one we discussed in the previous section. Let $\ell = \lfloor \log |se_2 \diamond e_1| \rfloor$. By the conditions of the maximiser, $|sx| \geq 2^\ell$. As in the previous section, we will find a vertex y in xs path that lies in $\mathcal{L}_\ell$. Again, the reader can verify that y is a D-close vertex to e_2 from s.

(2) $|se_1| < |se_2 \diamond e_1|$:This is the most complex case for us. Unlike the previous case, we cannot find a D-close vertex using one invocation of the maximiser. The reader will see that we will use different maximisers two more times. We will show that vertices returned by these maximisers will slowly lead us to the desired D-close vertex.

1. The first maximiser (stated above) will give us a vertex y, which is **close** to e_1. Note that we want to find a D-close vertex to e_2.
2. The second maximiser will give us a vertex $p \in P$ that has one important property: a does not lie in the trapezoid of $P[p, t]$. This is the only place in our paper where we use the concept of the trapezoid.
3. Once we have p, we can find the D-close vertex to e_2 in one call to an appropriate maximiser. This D-close vertex, say y, also satisfies the above property. That is, a does not lie in the trapezoid of $P[y, t]$.

Before we move further, we describe another issue. As stated above, in this case, we will find two vertices, y and p. After we found, say, y, it may be the case that the primary path yt passes through e_2 and the secondary path passes through e_1. This is actually a good case for us as we recurse our operations in the Section 5.1 to find a D-close vertex to e_2 from y. In the rest of the section, we will assume the following:

▶ **Assumption 17.** *All primary paths contain e_1, and the secondary paths contain e_2.*

Together with this assumption, we also show that, while recursing on P having any vertex as the source, if $e_1 = (a, b)$ lies on the primary path, then a is closer to the source. Similarly, if $e_2 = (c, d)$ lies on secondary path, then c is closer to the source. Remember that we had assumed this for the source s at the start of this section. This section will show that this assumption remains valid for all primary and secondary paths.

▶ **Lemma 18.** *Let $e_1 = (a, b)$ be the edge on the primary path st where a is closer to s. Similarly, let $e_2 = (c, d)$ lies on $st \diamond e_1$ and c is close to s in $st \diamond e_1$. Also, let p be a vertex on path $P[s, t]$. If e_1 lies on the primary path pt, then a is closer to p. Similarly, if e_2 lies on $pt \diamond e_1$, then c is closer to p on $st \diamond e_1$.*

Proof. Since a is closer to s in st path, b is closer to t than a. If b is closer to p on the pt path, then we can directly use the bt subpath of the st path and get the shortest pt primary path without using the edge e_1. This violates the condition of the claim. Hence, a is closer to p.

The proof for the second part of the claim is similar. ◀

With this, we have addressed all the small issues. Let us now look at our three maximisers.

First maximiser. Let us first see how to find a close vertex to e_1 (See Figure 8). Let x be a vertex returned by the maximiser such that x lies on the first segment of P.

Let $\ell = \lfloor \log |se_1| \rfloor$. By the first condition of the maximiser, $|sx| \geq 2^\ell$. We now find the first vertex of $\mathcal{L}_\ell$ in sx path from s whose distance is at most 2^ℓ. Using Lemma 12, such a vertex, say $y \in \mathcal{L}_\ell$, exists with a high probability. By Assumption 17, the primary path yt passes through e_1. We will now show that $|ye_1| = O(2^\ell)$. To this end, we will find a path from y to e_1 that avoids $\{e_1, e_2\}$ and has length $O(2^\ell)$. Consider the subpath ys of sx

concatenated with se_1. Since sx is intact from failures $\{e_1, e_2\}$, so is sy. And by construction, $|sy| = O(2^\ell)$. Similarly, se_1 avoids $\{e_1, e_2\}$ and is also of length $O(2^\ell)$. Thus, $|ye_1| = O(2^\ell)$. By definition, y is close to e_1.

Second maximiser. We will now show how to use y to find the vertex p. To this end, we define another maximiser. Let $dist[0] = |ye_1|$. The conditions of the maximiser are as follows:

$$\mathbb{D}_{0\alpha}(y, t, dist, clean) = \arg\max_{F \in E^2} \{dist[0] \text{ distance from } s \text{ on the primary path and } \textbf{secondary} \text{ path is intact from failures } F \text{ and other conditions based on } \alpha \text{ from the } t \text{ side.}\} \quad (6)$$

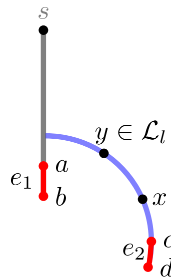

Figure 8 y is close to e_1.

Let p be the endpoint of an edge returned by the maximiser such that p lies on the first segment of $P[y, t]$. Before we move further, let us take a slight detour in our discussion. Till now, we have seen that if a vertex returned by the maximiser does not lie on the first segment, then it may be used as a clean vertex (for example, maximiser 4 and 5). However, we will not use p as a clean vertex, even if it satisfies the definition of the s-clean vertex. This is done to conserve space. Since this is an important technical point, we add this as a remark which will be used crucially when we calculate the space taken by the maximiser.

▶ Remark 19. Any vertex p returned as the output of the maximiser (6) will not be used as a clean vertex in the subsequent call to another maximiser.

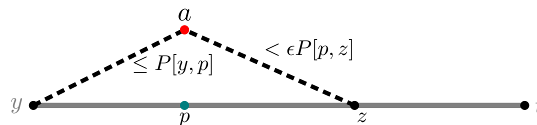

Figure 9 a does not lie in the trapezoid of $P[p, t]$.

The above remark implies that once we have found p on the first segment of path P, we will return $|sp| + \textsc{Query}(p, t, \{e_1, e_2\}, 2)$. Thus, we will recurse our algorithm with p as the source. But p has one special property that s did not have: the trapezoid of the path $P[p, t]$ (that is $pt \diamond \{e_1, e_2\}$), does not contain a – we now prove this claim.

▶ **Lemma 20.** *Let p be a vertex returned by maximiser (6) such that p lies on the first segment of $P[y, t]$. Then, a does not lie in the trapezoid of the path $P[p, t]$. Or by definition of trapezoid, for all $z \in P[p, t]$, $|za \diamond \{e_1, e_2\}| > \epsilon \min\{|P[p, z]|, |P[z, t]|\}$.*

Proof. For contradiction, let a lie in the trapezoid of the subpath $P[p,t]$. Then, there exists a vertex z on $P[p,t]$ such that $|za \diamond \{e_1, e_2\}| \leq \epsilon \min\{|P[p,z]|, |P[z,t]|\} \leq \epsilon |P[p,z]|$. Now we will find a path from y to z that avoids all the faults and is less than $P[y,z]$. Consider the path ya concatenated with $az \diamond \{e_1, e_2\}$ (See Figure 9). By Assumption 17, ya does not contain any faults. Thus, the above path avoids both e_1 and e_2.

Let us now calculate the length of this path.

$$\begin{aligned} |ya| + |az \diamond \{e_1, e_2\}| &= |ya| + |za \diamond \{e_1, e_2\}| \leq |yp| + |za \diamond \{e_1, e_2\}| \\ &\qquad \text{(By definition of the maximiser, } |yp| \geq |ya|) \\ &\leq |P[y,p]| + |za \diamond \{e_1, e_2\}| \qquad \text{(as } P[y,p] \geq |yp|) \\ &\leq |P[y,p]| + \epsilon |P[p,z]| < |P[y,p]| + |P[p,z]| = |P[y,z]| \end{aligned}$$

This is a contradiction as $P[y,z]$ is the shortest path from y to z avoiding both faults. So, our assumption was wrong. So, for any $z \in P[p,t]$, $|za \diamond \{e_1, e_2\}| > \epsilon |P[p,z]|$. In other words, a lies outside trapezoid of $P[p,z]$. ◀

We will now use the vertex p to find a D-close vertex to e_2. To this end, we need another useful claim to design the maximiser.

▶ **Lemma 21.** *Let z be the least common ancestor of e_1 and e_2 in the path pe_1 and $pe_2 \diamond e_1$. Let i be the greatest integer such that $(1+\epsilon)^i$ is less than $|pa|$ on the primary path pt. Then the length of the path $|pz|$ on the primary path is $\leq (1+\epsilon)^i$. See Figure 10 for an illustration of the lemma.*

Proof. Assume for contradiction that $|pz| > (1+\epsilon)^i$. Then,

$$|za \diamond \{e_1, e_2\}| = |za| = |pa| - |pz| \leq (1+\epsilon)^{(i+1)} - (1+\epsilon)^i = \epsilon(1+\epsilon)^i \leq \epsilon |pz| = \epsilon |P[p,z]|$$

This violates Lemma 20. Hence, our assumption was wrong. ◀

Last maximiser. We are now ready to define our last maximiser. First we will find the largest i such that $(1+\epsilon)^i \leq |pe_1|$. Also, we set $dist[0] = 2^{\lfloor \log |pe_2 \diamond e_1| \rfloor}$. We will use the following conditions in the maximiser:

$$\mathbb{D}_{0\alpha}(p, t, dist, clean) = \arg\max_{F \in E^2} \{\ (1+\epsilon)^i \text{ distance from } p \text{ on the primary path is intact from } F\ ;\ dist[0] \text{ distance from } p \text{ on the secondary path is intact from } F; \text{ Other conditions based on } \alpha \text{ from } t \text{ side. } \} \tag{7}$$

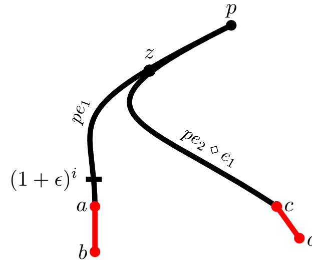

Figure 10 Illustration for Lemma 21.

Let x be an endpoint of an edge returned by the maximiser such that it lies in the first segment of $P[p,t]$. We first show that x cannot lie on the primary path pt and should necessarily lie on the secondary path.

▶ **Lemma 22.** *Let x be an endpoint of an edge returned by the maximiser (7) such that x lies in the first segment of $P[p,t]$. Let z be the least common ancestor of e_1 and e_2 in the path pe_1 and $pe_2 \diamond e_1$. Then, x cannot lie in pz, but it lies on $ze_2 \diamond e_1$.*

Proof. Let i be the greatest integer such that $(1+\epsilon)^i$ is less than $|pa|$ on the primary path pt. Using Lemma 21, we know that $|pz| \le (1+\epsilon)^i$. One of the conditions of the maximiser is that $(1+\epsilon)^i$ distance from p on the primary path remains intact. Thus, x cannot lie on sz. Since $x \in P$ and the detour starts on the secondary path, x lies on $ze_2 \diamond e_1$. ◀

Let $\{e_1^*, e_2^*\}$ be the edges the maximiser returns. Without loss of generality, let us assume that e_1^* lies on the primary path. Using the above lemma, x cannot be an endpoint of e_1^*. This implies that x lies on $pt \diamond e_1^*$ or the secondary path. However, we have a condition for the secondary path in our maximiser.

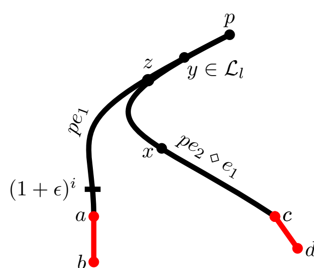

Figure 11 The vertex y is D-close.

Let $\ell = \lfloor \log |pe_2 \diamond e_1| \rfloor$. One of the conditions of the maximiser is that 2^ℓ distance from s on the secondary path is intact from failures. Thus, $|px \diamond e_1^*| \ge 2^\ell$. But x lies on the first segment of $P[p,t]$. So $|px \diamond e_1^*| = |px| \ge 2^\ell$. Now, using x, we will find a vertex that is *close* to e_2. Using Lemma 12, with a high probability, we will find a vertex, say y, in path px from x that lies in $\mathcal{L}_\ell$ (See Figure 11). Since $|pe_2 \diamond e_1| = O(2^\ell)$, even $|ye_2 \diamond e_1| = O(2^\ell)$. Thus, y is close to e_2. We will now show that it is D-close to e_2. To this end, we note that the detour of $P[p,t]$ starts on the secondary path after the vertex x (as x is intact from failures). Thus, the detour also starts after the vertex y on the path $P[p,t]$. Thus, y is a D-close vertex to e_2.

Remember that p satisfied Lemma 20. We claim that once a is outside the trapezoid of path $P[p,z]$, it is outside the trapezoid of any of its suffix paths. Specifically, for $P[y,t]$, a lies outside its trapezoid. The reader can verify that Lemma 20 holds even if we replace p with y. Thus, Lemma 20 and 21 hold even for $p[y,t]$ path.

6 From D-close vertex to a clean vertex

Let us describe the setting first. In the previous section, we have found a D-close vertex from the source s. We will abuse notation and assume that the vertex s itself is a D-close vertex and $s \in \mathcal{L}_\ell$. We now show how to find a clean vertex using s. There are two cases, as in the previous section.

6.1 The detour starts on the primary path

In this case, we will use another maximiser similar to maximiser (4). First, we set $dist[0] = |se_1|$. The conditions of the maximiser are as follows:

$$\mathbb{D}_{1\alpha}(s, t, dist, clean) = \arg\max_{F \in E^2}\{dist[0] \text{ distance from } s \text{ on primary path is intact from } F \text{ ;Other conditions based on } \alpha \text{ from the } t \text{ side.}\} \quad (8)$$

Let x be an endpoint of an edge returned by the maximiser such that x lies on the first segment of P. We will now show that x is an s-clean vertex. We claim that Lemma 16 also applies for maximiser (8). Indeed, in this lemma, we are only using the fact that the detour starts on the primary path – which is valid for this case too. Now, since x lies on the first segment and x cannot lie on the se_1 path (by the condition of the maximiser), x cannot lie on the st path. Hence, using Lemma 16, x is an s-clean vertex.

6.2 The detour starts on the secondary path

Let us again reiterate the setting that leads us to a D-close vertex at the end of Section 5.2. Again, we will abuse notation and assume that s is a D-close vertex to e_2. Also, Lemma 20 holds for s. So, a is outside the trapezoid of $P[s, t]$ path. Using this property, we can say that Lemma 21 also holds for s. Let i be the largest integer such that $(1 + \epsilon)^i \leq |se_1|$. Let $dist[0] = |se_2 \diamond e_1|$. The conditions of our maximiser are as follows (it is the same as maximiser (7)):

$$\mathbb{D}_{1\alpha}(s, t, dist, clean) = \arg\max_{F \in E^2}\{ (1 + \epsilon)^i \text{ distance from } s \text{ on primary path and } dist[0] \text{ distance from } s \text{ on secondary path is intact from } F \text{ ; other conditions based on } \alpha \text{ from } t \text{ side.} \quad (9)$$

Let x be an endpoint of an edge returned by the maximiser such that x lies on the first segment of P. We will now show that x is an s-clean vertex. To prove our claim, we will show that neither $V(e_1)$ nor $V(e_2)$ intersects with $T_s(x)$ as we have already assumed that sx is intact from failures $\{e_1, e_2\}$.

The proof that $V(e_1) \cap T_s(x) = \emptyset$ follows from arguments similar to Lemma 22. Let us first show that $V(e_1) \cap T_s(x) = \emptyset$. Let z be the least common ancestor of e_1 and e_2 in the paths se_1 and $se_2 \diamond e_1$. Since s satisfies Lemma 21, $|sz| \leq (1 + \epsilon)^i$. The first condition of the maximiser is that $(1 + \epsilon)^i$ distance from s on the primary path is intact from failures. Hence, x cannot lie on the sz path. It must lie on $ze_2 \diamond e_1$. Thus, $T_s(a) \cap T_s(x) = \emptyset$. So, $V(e_1) \cap T_s(x) = \emptyset$. Let $\{e_1^*, e_2^*\}$ be the edges returned by the maximiser. Without loss of generality, let us assume that e_1^* lies on the primary path. Using similar arguments like Lemma 22, x is not an endpoint of e_1^* as x does not lie on the primary path. This implies that x lies on $st \diamond e_1^*$ or the secondary path. However, we have a condition for the secondary path in our maximiser.

Our second condition is that $|se_2 \diamond e_1|$ distance from s on the secondary path is intact from failures. Thus, $|sx| \geq |se_2 \diamond e_1|$. Or, $|sx| \geq |se_2| = |sc|$. Thus, $T_s(x)$ cannot contain the vertex c.

We will now show that $d \notin T_s(x)$. Our argument here is the same as in the proof of Lemma 16. We repeat the arguments here for the sake of completeness. If $d \in T_s(x)$, then since $c \notin T_s(x)$, there exists some c' such that $c'd$ is an edge of $T_s(x)$. Consider the path from s to x concatenated with the path from x to d in $T_s(x)$. This concatenated path represents

the shortest path from s to d, and it also avoids e_1. Additionally, it should be noted that T_s was constructed by adding small weights to each edge. Therefore, $|sd \diamond e_1| = |sd|$ is unique and does not pass through e_2. Furthermore, it should be remembered that we found $st \diamond e_1$ in the same weighted graph, and thus $st \diamond e_1$ is also unique and passes through e_2. However, according to the property of shortest paths, $sd \diamond e_1$ should be a subpath of $st \diamond e_1$. This leads to a contradiction, implying that $d \notin T_s(x)$. Thus, once we find a D-close vertex, the next vertex we can find on the first segment of P has to be an s-clean vertex.

7 From a clean vertex to an intermediate vertex

We will assume that we have found an s-clean vertex, say p, such that the path P passes through p using the process discussed in the previous section. In this section, we will show how to use a maximiser to find an intermediate vertex. First set $clean[0] = p$. The conditions of the maximiser are as follows:

$$\mathbb{D}_{2\alpha}(s, t, dist, clean) = \arg \max_{F \in E^2} \{clean[0] \text{ is } s\text{-clean from } F, \text{ other conditions based on } \alpha \text{ from } t \text{ side.}\} \tag{10}$$

We claim that none of the endpoints of the edges returned by the maximiser is in the first segment of P. For contradiction, let x be the vertex returned by the maximiser, which is in the first segment of P. Then, the path sx remains intact from faults $\{e_1, e_2\}$. Since p is s-clean, the path sp is intact from faults $\{e_1, e_2\}$. Also, both p and x lie on the path P. Therefore, either sp contains x or sx contains p. However, if $x \in sp$, it contradicts our condition in the maximiser that sp is intact from faults. Thus, this possibility cannot arise. Similarly, if sx contains p, then $x \in T_s(p)$. However, this violates the condition in the maximiser that $T_s(p)$ is intact from faults. As both possibilities lead to a contradiction, our assumption is false. Hence, x cannot lie in the first segment of P. Now, if x also avoids the last segment, x becomes an intermediate vertex, and we are done.

We discuss the space of our oracle, time taken by the query algorithm and the case when both the faults lie on the primary path in Appendix C.

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.
2. Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.
3. 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.
4. 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. ACM, 2009.
5. 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.

6 Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.
7 Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. Comput.*, 37(5):1299–1318, 2008.
8 Dipan Dey and Manoj Gupta. Nearly optimal fault tolerant distance oracle. *arXiv preprint arXiv:2402.12832*, 2024.
9 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.
10 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.
11 François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303, 2014.
12 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.
13 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.
14 Adam Karczmarz and Piotr Sankowski. Sensitivity and dynamic distance oracles via generic matrices and frobenius form. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, 2023.
15 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.
16 Andrew James Stothers. On the complexity of matrix multiplication, 2010.
17 Jan van den Brand and Thatchaphol Saranurak. Sensitive distance and reachability oracles for large batch updates. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 424–435. IEEE Computer Society, 2019.
18 Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Trans. Algorithms*, 9(2), March 2013. doi:10.1145/2438645.2438646.
19 Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 887–898, 2012.

A Space required by our data-structures

Consider one of the maximisers of [10].

$$\mathbb{D}_{01}(s,t,x,y) = \\ \text{argmax}_{\{e_1,e_2\}\in E^2}\ \{sx \text{ is intact from faults } \{e_1,e_2\} \text{ and } y \text{ is } t\text{-clean from } \{e_1,e_2\}\ \}$$

To calculate the size of the data structure, we can consider two terms: the first term represents combinations of the source vertex s and the vertex x (referred to as a helper vertex in [10]). The number of combinations of sources and helper vertices can be $O(n^2)$. Similarly, the number of combinations of the destinations (such as t) and the helper vertices (such as y) can also be $O(n^2)$. Multiplying these two terms gives us a total space of $O(n^4)$. We now formally define this notion which will help us to calculate the space taken by our algorithm.

▶ **Definition 23** (Space requirement from the source side). *Let us assume that the maximiser function has k parameters completely dependent on the source. The space required from the source side is the number of possible combinations of these k parameters and the source. Similarly, we can define the space requirement from the destination side.*

We will primarily determine the required space from the source side. In most cases, this space will be $\tilde{O}(n)$. The calculation is symmetrical from the destination side, resulting in a total space bound of $\tilde{O}(n^2)$. However, this approach is only possible for a few maximisers. Consequently, we divide our maximisers into two groups. The space taken by the first group can be calculated using the approach mentioned above, while calculating the space requirement of the second group requires more complex techniques. Here is the partition:

1. $\mathbb{D}_{\alpha\beta}$ where both $\alpha, \beta \in [0,1]$ and
2. $\mathbb{D}_{\alpha\beta}$ where α and/or β is equal to 2.

For the maximisers in group (1), the values of $clean[0]$ and $clean[1]$ are not set. These maximisers are used when neither an s-clean nor a t-clean vertex has been found. On the other hand, for the maximisers in the group (2), the value of $clean[0]$ and/or $clean[1]$ is set. Let us first bound the size of the group (1) maximisers.

A.1 Group 1 maximisers

There are two kinds of maximisers in this group: the first one in which we have not found a D-close vertex from s and the second in which we have found the D-close vertex. We will bound the size of these maximisers separately.

1. $\mathbb{D}_{0\alpha}$ where $\alpha \in \{0,1\}$

   There are four maximisers of type $\mathbb{D}_{0\alpha}$, namely maximiser (4), (5), (6), and (7). Among these maximisers, three of them have at most $\log n$ values set for $dist[0]$, where $dist[0]$ is:

   - $2^{\lfloor \log |se_1| \rfloor}$ in maximiser (4)
   - $2^{\lfloor \log(\min |se_1|, |se_2 \diamond e_1|) \rfloor}$ in maximiser (5)
   - $2^{\lfloor \log |pe_2| \rfloor}$ in maximiser (7)

   For maximisers (4) and (5), $dist[0]$ has $\log n$ possibilities and the number of sources can be at most n. Thus, the space these maximisers take from the source side is $\tilde{O}(n)$. In the case of maximiser (7), there is an additional condition that the $(1+\epsilon)^i$ distance on the primary path remains intact from failures. Again, i can take $\log n$ values (assuming ϵ is a constant). Hence, the space taken by maximiser (7) from the source side is also $\tilde{O}(n)$.

   Now, let us consider maximiser (6). In this maximiser, y is a *close* vertex to e_1. Let us assume that $y \in \mathcal{L}_\ell$. We set $dist[0] = |ye_1|$. Since y is close to e_1, the number of possible values for $dist[0]$ is $O(2^\ell)$. Thus, the total space taken by this maximiser from the source side is $\sum_{\ell=0}^{\log n} \tilde{O}\left(\frac{n}{2^\ell} \times 2^\ell\right) = \tilde{O}(n)$.
2. $\mathbb{D}_{1\alpha}$ where $\alpha \in \{0,1\}$.

   There are four maximisers of type $\mathbb{D}_{1\alpha}$, namely maximiser (8) and (9). We use these maximisers after we find a D-close vertex from the source side. For simplicity, let us assume that the source vertex s itself is D-close. We further assume that $s \in \mathcal{L}_\ell$. Since s is D-close to the fault, we can set $O(2^\ell)$ different values for $dist[0]$ in maximiser (8) and (9). Additionally, similar to maximiser (6), we can set at most $\log n$ values for i in maximiser (9). Thus, the space these two maximisers take from the source side is $\tilde{O}(n)$.

Thus, the number of maximisers of type $\mathbb{D}_{00}, \mathbb{D}_{01}, \mathbb{D}_{10}$ and $\mathbb{D}_{11}$ is $\tilde{O}(n^2)$.

A.2 Group 2 maximisers

In a Group 2 maximiser, we set the values of $clean[0]$ and/or $clean[1]$. One might assume that there could be $O(n)$ possibilities for $clean[0]$ and $clean[1]$, implying that the size of Group 2 maximisers should be strictly greater than $\tilde{O}(n^2)$. However, this is not the case. Firstly, we claim that the number of possible vertices in $clean[0]$ is limited. For instance, let us consider the maximiser $\mathbb{D}_{20}(s, t, dist, clean)$. In this maximiser, we set the value of an s-clean vertex as $clean[0]$. However, where does this s-clean vertex come from? A keen reader can observe that we must have found this s-clean vertex in a previous call to either the maximiser $\mathbb{D}_{00}(s, t, \cdot, \cdot)$ or $\mathbb{D}_{10}(s, t, \cdot, \cdot)$.

We have already bounded the size of $\mathbb{D}_{00}$ and $\mathbb{D}_{10}$ maximisers, and each maximiser returns at most four vertices. Thus, we conclude that there are at most four possible values for $clean[0]$ in $\mathbb{D}_{20}(s, t, dist, clean)$ for each maximiser of form $\mathbb{D}_{00}(s, t, \cdot, \cdot)$ or $\mathbb{D}_{10}(s, t, \cdot, \cdot)$. Thus, the number of possible vertices in $clean[0]$ is $\tilde{O}(n^2)$.

Using the same strategy, we can bound the space taken by Group 2 maximisers. There is only one Group 2 maximiser, that is, maximiser (10). This maximiser can be of three types: $\mathbb{D}_{20}, \mathbb{D}_{21}$ and $\mathbb{D}_{22}$. We will individually bound the space for each type.

1. $\mathbb{D}_{20}$
   In this case, we have already found an s-clean vertex from the source side. As mentioned earlier, this s-clean vertex must be the output of a maximiser, specifically of type $\mathbb{D}_{00}$ or $\mathbb{D}_{10}$. Let's assume that we have found an s-clean vertex x from the maximiser $\mathbb{D}_{00}(s, t, dist', clean')$. Two changes are made in the subsequent call to the maximiser $\mathbb{D}_{20}(s, t, dist, clean)$. First, we set $bits[0] = 2$, which renders $dist[0]$ irrelevant (as in maximiser (10), $dist[0]$ is not used in any condition). The main change is setting $clean[0]$ to x. Since each maximiser outputs two edges or four vertices, for each maximiser of type $\mathbb{D}_{00}$ or $\mathbb{D}_{10}$, at most four maximisers of type $\mathbb{D}_{20}$ are created. Hence, the total size of $\mathbb{D}_{20}$ is $\tilde{O}(n^2)$. Similarly, the size of $\mathbb{D}_{02}$ is also $\tilde{O}(n^2)$.
2. $\mathbb{D}_{21}$
   Once again, let us consider the scenario that occurred in our algorithm just before utilising this data structure. At that point, we either discovered an s-clean vertex or identified a D-close vertex from t. Let us focus on the more straightforward case we have already addressed earlier.
   a. We have found an s-clean vertex.
      The s-clean vertex obtained must have been the output of either the maximiser $\mathbb{D}_{01}$ or $\mathbb{D}_{11}$. Similar to the previous case, we assert that the overall size of such a data structure is $\tilde{O}(n^2)$.
   b. We have found a D-close vertex from t.
      This D-close vertex must be the output of the maximiser $\mathbb{D}_{20}$. Let t be a D-close vertex found using the maximiser $\mathbb{D}_{20}(s, t', dist', clean')$. After finding t, our algorithm uses the maximiser $\mathbb{D}_{21}(s, t, dist, clean)$. The pair $s, clean[0]$ remains unchanged between $\mathbb{D}_{20}$ and $\mathbb{D}_{21}$. The values of $dist[0]$ and $dist'[0]$ are not used in the conditions of maximisers $\mathbb{D}_{21}$ and $\mathbb{D}_{20}$, respectively, so they remain unchanged.
      The only change is that t' in $\mathbb{D}_{20}$ is replaced by the D-close vertex t, and $dist[1]$ is set to a suitable value. Let $t \in \mathcal{L}_\ell$. The number of possible values for $dist[1]$ is $O(2^\ell)$. Thus, the total space taken from the destination side in the maximiser $\mathbb{D}_{21}$ is $\sum_{\ell=0}^{\log n} \frac{n}{2^\ell} 2^\ell = \tilde{O}(n)$.
      We will now examine the total space from the source side and show it as $\tilde{O}(n)$. As stated above, there is no change in s and $clean[0]$ between the maximisers $\mathbb{D}_{20}$ and $\mathbb{D}_{21}$. Thus, the space taken by the maximiser $\mathbb{D}_{21}$ from the source side is the same as

the space taken by the maximiser $\mathbb{D}_{20}$ from the source side. We now bound the space taken by $\mathbb{D}_{20}$ from the source side.

We use the maximiser $\mathbb{D}_{20}$ after we obtain an s-clean vertex either from $\mathbb{D}_{00}$ or $\mathbb{D}_{10}$. We look at these two transitions to bound the space of $\mathbb{D}_{20}$ from the source side. Let us go over these two transitions separately:

- The s-clean vertex in the maximiser $\mathbb{D}_{20}$ is obtained from the maximiser $\mathbb{D}_{00}$.
  Consider the maximiser $\mathbb{D}_{00}(s, t, dist, clean)$. We assume that we have found an s-clean vertex from this maximiser, which is then used in the subsequent call to maximiser $\mathbb{D}_{20}$. By applying Remark 19, this s-clean vertex cannot be the output of maximiser (6).
  Therefore, the maximiser $\mathbb{D}_{00}$ can fall into types (4), (5), and (7). In these maximisers, there can be at most $\log n$ different values for $dist[0]$ and $dist[1]$. Consequently, at most $O(\log^2 n)$ clean vertices can serve as the output of the maximiser $\mathbb{D}_{00}$. As a result, the total number of possible combinations of source and $clean[0]$ in the maximiser $\mathbb{D}_{20}$ is bounded by $O(n \log^2 n)$. Hence, the space required from the source side in the maximiser $\mathbb{D}_{20}$ is $\tilde{O}(n)$.
- The s-clean vertex in the maximiser $\mathbb{D}_{20}$ is obtained form the maximiser $\mathbb{D}_{10}$.
  Consider the maximiser $\mathbb{D}_{10}(s, t, dist, clean)$. We assume that we have found an s-clean vertex from this maximiser, which is then used in the subsequent call to the maximiser $\mathbb{D}_{20}$. Again, by using Remark 19, this s-clean vertex cannot be the output of the maximiser 6.
  If the maximiser $\mathbb{D}_{10}$ falls into types (4), (5), or (7), then there can be at most $\log n$ different values of $dist[1]$. By the definition of $\mathbb{D}_{10}$, s is a D-close vertex from the source. Let us assume that $s \in \mathcal{L}_\ell$. Consequently, the number of possible different values of $dist[0]$ is at most $O(2^\ell)$.
  As a result, at most $O(2^\ell \log n)$ clean vertices can serve as the output of the maximiser $\mathbb{D}_{10}$. Thus, the total number of possible combinations of source and $clean[0]$ in $\mathbb{D}_{20}$ is $\sum_{\ell=0}^{\log n} O\left(\frac{n}{2^\ell} 2^\ell \log n\right)$. Therefore, the space required from the source side in $\mathbb{D}_{20}$ is $\tilde{O}(n)$.

Thus, the total space taken by $\mathbb{D}_{21}$ is $\tilde{O}(n^2)$. The same argument also applies to $\mathbb{D}_{12}$.

3. $\mathbb{D}_{22}$
   This data structure is invoked when we obtain either an s-clean or a t-clean vertex from $\mathbb{D}_{20}$, $\mathbb{D}_{21}$, $\mathbb{D}_{02}$, or $\mathbb{D}_{21}$. All of these data structures have a size of $\tilde{O}(n^2)$. As each maximiser outputs two edges or four vertices, the size of $\mathbb{D}_{22}$ is also $\tilde{O}(n^2)$.

B Time taken by the Query algorithm

First, we note that it takes $O(1)$ time to determine all the parameters of the maximiser. From the source side, to set $dist[0]$, we need to find $|se_1|$ and $|se_2 \diamond e_1|$. Since we do not know whether the detour starts or ends on the primary or secondary path, we exhaustively check all possibilities. There are only four possible cases:

1. The detour starts and ends on the primary path.
2. The detour starts on the secondary path but ends on the primary path.
3. The detour starts on the primary path but ends on the secondary path.
4. The detour starts and ends on the secondary path.

Let us first consider the most straightforward case: when the detour starts and ends on the primary path. Discussing the running time in the context of the flowchart presented in Figure 5 is easier. After the invocation of the first maximiser, namely maximiser (4), in

the worst case, we can obtain a D-close vertex from s. The second invocation may give a D-close vertex from t, and so on. We claim that we must have found the intermediate vertex after four invocations of maximisers. Since each maximiser returns two edges or four vertices, there are at most 16 vertices to be processed. This can be done in $O(1)$ time. Thus, we obtain the intermediate vertex in $O(1)$ time.

The analysis is similar for the second case: when the detour starts on the secondary path but ends on the primary path. In this case, finding a D-close vertex from the source vertex itself may require three invocations of maximisers (see Section 5.2). Thus, the total number of calls to the maximiser increases to six. Even in this case, the running time is $O(1)$.

The last two cases are similar to the previous two cases. Since we do not know which case we are in, we run our algorithm for all four cases. Each case takes $O(1)$ time, after which we obtain $O(1)$ candidates for the intermediate vertices. Simultaneously, we always maintain an upper bound on the length of $P = |st \diamond \{e_1, e_2\}|$. Using Lemma 15, if the edges returned by any of our maximisers do not intersect with P, we can update our upper bound on the size of P (represented by the value L returned by the HitSet algorithm). Therefore, our HitSet algorithm returns an upper bound on P and a set H of $O(1)$ candidates for the intermediate vertices. Following the analysis in Section 2, the running time of our algorithm is $O(1)$.

C Both the faults are on the primary path

Let us first define the setting in this easy case when both the faulty edges lie on the primary path. We want to find the length of $P = st \diamond \{e_1, e_2\}$ where $e_1 = (a, b)$ and $e_2 = (c, d)$ lie on the primary path. We further assume that a is closer to s and d is closer to t.

In Section 4, we designed an algorithm for the case when the faults were on the primary as well as the secondary path. One of the corner cases in that section was as follows: We find a D-close vertex, say y, such that the primary path yt contains both faults. Thus, we jump from one case to the other. We will now show that we cannot jump in the reverse direction. To this end, we state the following lemma without proof. The proof of this lemma is trivial.

▶ **Lemma 24.** *If the path st has both the faults, then for any $y \in se_1$, both the faults are in the primary path yt. Similarly, for $y \in te_2$ path, both the faults are in the primary path sy.*

In the case we are dealing with in this section, the reader will see that if we find a D-close vertex, say p, from s, then it will lie on se_1 path. Thus, using the above lemma, we will remain in the case where both the faults lie on the primary path pt. Thus, we will not jump from this case to any other case. Let us now discuss how we handle this case.

The reader will see that this case is nearly similar to the case described in Section 5.1. Indeed, it is necessarily the case here that the detour starts and ends on the primary path. Our first maximiser is the same as the maximiser (4). We set $dist[0] = 2^{\log \lfloor |se_1| \rfloor}$. The conditions of the maximiser are as follows:

$$\mathbb{D}_{0\alpha}(s, t, dist, clean) = \arg\max_{F \in E^2} \left\{ dist[0] \text{ distance from } s \text{ on the primary path is intact from } F \text{ and Other conditions based on } \alpha \text{ from the } t \text{ side.} \right\} \quad (11)$$

The only difference between maximiser (11) and maximiser (4) is the common condition of (3). For our case, the common condition is as follows:

Common Condition:

A pair of edges (e, e') will be considered in the maximiser only if $e, e' \in st$ (12)

Let x be an endpoint of an edge returned by the maximiser such that it lies on the first segment of P. Now, like Lemma 16, we claim that either x lies on the st path or is s-clean. The proof of this is easier than the proof of Lemma 16 but follows similar arguments.

▶ **Lemma 25** (Same as Lemma 16). *Let x be an endpoint of an edge returned by the maximiser $\mathbb{D}_{0\alpha}$ such that sx lies on the first segment of P. If x does not lie on the st path, then x is an s-clean vertex.*

Proof. Let us assume that $x \notin st$. By the condition in the lemma, sx is intact from failures. Thus, to show that x is an s-clean vertex, we only need to show that the vertices in $V(e_1)$ and $V(e_2)$ do not lie in $T_s(x)$.

Since x is not in the st path and hence not in the sa path, the two subtrees $T_s(x)$ and $T_s(a)$ are disjoint. Here, $e_1, e_2 \in T_s(a)$ (as both the edges are on the primary path itself). So, $(V(e_1) \cup V(e_2)) \cap T_s(x) = \emptyset$. ◀

If x lies on the st path then, similar to Section 5.1, we find a D-close vertex to e_1 using x, say y. Here, note that, by construction, y lies on se_1 path. Using Lemma 24, the primary path yt also contains faults $\{e_1, e_2\}$. Then, using a maximiser similar to the one used in Section 6.1, we can find an s-clean vertex. Similarly, we get an intermediate vertex using a maximiser similar to the one used in Section 7. Since our arguments are similar to the arguments in Section 6.1 and Section 7, we do not repeat them here. The space and the time taken in this case also can be bounded using the arguments presented in Appendix A and Appendix B, respectively. Thus, we have proven the main result of the paper, that is Theorem 1.