Near Optimal Algorithm for Fault Tolerant Distance Oracle and Single Source Replacement Path Problem

Dipan Dey ✉
IIT Gandhinagar, India

Manoj Gupta ✉ 🏠
IIT Gandhinagar, India

── **Abstract** ───────────────

In a graph G with a source s, we design a distance oracle that can answer the following query: $\text{QUERY}(s, t, e)$ – find the length of shortest path from a fixed source s to any destination vertex t while avoiding any edge e. We design a deterministic algorithm that builds such an oracle in $\widetilde{O}(m\sqrt{n})$ time[1]. Our oracle uses $\widetilde{O}(n\sqrt{n})$ space and can answer queries in $\widetilde{O}(1)$ time. Our oracle is an improvement of the work of Bilò et al. (ESA 2021) in the preprocessing time, which constructs the first deterministic oracle for this problem in $\widetilde{O}(m\sqrt{n} + n^2)$ time.

Using our distance oracle, we also solve the *single source replacement path problem* (SSRP problem). Chechik and Cohen (SODA 2019) designed a randomized combinatorial algorithm to solve the SSRP problem. The running time of their algorithm is $\widetilde{O}(m\sqrt{n} + n^2)$. In this paper, we show that the SSRP problem can be solved in $\widetilde{O}(m\sqrt{n} + |\mathcal{R}|)$ time, where $\mathcal{R}$ is the output set of the SSRP problem in G. Our SSRP algorithm is optimal (upto polylogarithmic factor) as there is a conditional lower bound of $\Omega(m\sqrt{n})$ for any combinatorial algorithm that solves this problem.

1 Introduction

Real-life graph networks are prone to failures, e.g., nodes or links can fail. Thus, algorithms developed for these networks must be resilient to failures. For example, there may be some edges or links which are not working in the network and we want to avoid them. In this paper, we present an algorithm to create an oracle for the single source shortest path problem in a fault-prone graph. Such algorithms are also called fault-tolerant algorithms.

Consider an undirected and unweighted graph G with a source s. We want to build an oracle that can find the length of shortest path from s to any other vertex in the presence of faulty edges – such an oracle is also called a *fault-tolerant distance oracle*. Formally,

▶ **Definition 1.** *A fault-tolerant distance oracle answers the following query in a graph G:*

$\text{QUERY}(s, t, F)$: *Find the length of shortest path from s to t avoiding the set F of edges.*

─────────────────────

[1] $\widetilde{O}()$ hides polylog n factor

The time it takes to answer a query is called the *query time*. If the query is always from a fixed source s and $|F| \leq f$, then the distance oracle is called a f-edge fault tolerant single source distance oracle, or $\textsc{Sdo}(f)$ in short. If all vertices can be sources, the oracle is called f-edge fault tolerant distance oracle, or $\textsc{Do}(f)$. We list some results related to distance oracles:

Demetrescu et al. [12] designed a $\textsc{Do}(1)$ with $\widetilde{O}(n^2)$ space and $O(1)$ query time. Bernstein and Karger[3] showed that this oracle can be built in $\widetilde{O}(mn)$ time. Pettie and Duan [15] extended the result of Demestrescu et al. to two faults. They designed a $\textsc{Do}(2)$ with $\widetilde{O}(n^2)$ space and $\widetilde{O}(1)$ query time. Gupta and Singh [20] designed a $\textsc{Sdo}(1)$ with $\widetilde{O}(n\sqrt{n})$ space and $\widetilde{O}(1)$ query time. Recently Bilò et al. [5] built the $\textsc{Sdo}(1)$ (described in [20]) in $\widetilde{O}(m\sqrt{n}+n^2)$ time. They also showed that, the lower bound of size for such an oracle is $O(n\sqrt{n})$ Many different aspects of distance oracles have been studied in literature [4, 3, 6, 9, 10, 14, 29]. In the line of k simple shortest paths the some important works are [27, 28, 2]

In this paper, we will focus our attention on building $\textsc{Sdo}(1)$. Due to Bilò et al. [5], the time to build $\textsc{Sdo}(1)$ is $\widetilde{O}(m\sqrt{n}+n^2)$. Chechik and Cohen [8] showed that, the first term in this running time is a conditional lower bound for $\textsc{Ssrp}$ problem. But it is not clear if the second term is necessary. In this paper, we build a $\textsc{Sdo}(1)$ in $\widetilde{O}(m\sqrt{n})$ time – this preprocessing algorithm has a better runtime than [5] for sparse graphs, which is state of the art for this problem till now. Using our $\textsc{Sdo}(1)$ data structure, we are able to reduce the runtime of the algorithm solving $\textsc{Ssrp}$ problem too. Our distance oracle is quite different from the distance oracle of Gupta and Singh [20] – though we use the main technical idea of [20] crucially in our paper too. The construction of this new oracle is the main technical result of this paper.

▶ **Theorem 2.** *For undirected, unweighted graphs there is a deterministic algorithm that can build a $\textsc{Sdo}(1)$ of size $\widetilde{O}(n\sqrt{n})$ and query time $\widetilde{O}(1)$ in $\widetilde{O}(m\sqrt{n})$ time.*

1.1   Application: Single Source Replacement Path Problem

Let us first look at the *replacement path problem*. In this problem, we are given a source s and a destination t. We assume that there is a unique shortest path from s to t, denoted by st.

▶ **Definition 3** (Replacement Path Problem). *Let s be a source and vertex t be the destination in G. For each $e \in st$ path, output the length of the shortest path from s to t avoiding e.*

The replacement path problem was first investigated due to its relation with auction theory [21, 24] and has been studied extensively. For an undirected graph with non-negative edge weights, the replacement path problem can be solved in $\widetilde{O}(m+n)$ time[22, 21, 23]. We look at the generalization of the replacement path problem – the single source replacement path problem.

▶ **Definition 4** (Ssrp problem). *Let s be a source in a graph G which is undirected and unweighted. For each vertex $t \in G$ and each $e \in st$ path, output the length of the shortest path from s to t avoiding e.*

Chechik and Cohen [8] designed a randomized combinatorial algorithm that solves the $\textsc{Ssrp}$ problem in $\widetilde{O}(m\sqrt{n}+n^2)$ time. They also showed a matching conditional lower bound via Boolean Matrix Multiplication.

▶ **Lemma 5** ([8])**.** *Let $\text{BMM}(n, m)$ be the time taken to multiply two $n \times n$ boolean matrices with a total of m ones. Under the assumption that any combinatorial algorithm for $\text{BMM}(n, m)$ requires $mn^{1-o(1)}$ time[2], any combinatorial algorithm for SSRP problem requires $\Omega(m\sqrt{n})$ time.*

It may seem that the algorithm of Chechik and Cohen [8] is nearly optimal. It is indeed the case if the output size is $O(n^2)$. However, for a low-diameter graph, this extra additive factor seems unnecessary. If the graph is dense ($m \geq n^{3/2}$), then the n^2 factor is subsumed by the first term $m\sqrt{n}$. Thus, when $m < n^{3/2}$ and the graph has a low diameter, can we improve the running time of the SSRP problem? For such a graph, the algorithm of Chechik and Cohen [8] is not optimal. Similar to [8], Gupta et al. [18] also designed an algorithm for the SSRP problem. Even this algorithm has the running time $\widetilde{O}(m\sqrt{n} + n^2)$ – though it uses an entirely different approach compared to [8]. Thus, the main question is: *can we remove this extra additive factor of n^2 from the running time of the SSRP problem?* In this paper, we design such an algorithm:

▶ **Theorem 6.** *There is a deterministic algorithm for SSRP problem with a running time of $\widetilde{O}(m\sqrt{n} + |\mathcal{R}|)$ where $|\mathcal{R}|$ is the output size of SSRP problem in G.*

In the above theorem, $|\mathcal{R}|$ is the output size, thus an implicit lower bound on the SSRP problem. Using Lemma 5, we conclude that our algorithm is nearly optimal up to a polylogarithmic factor.

To build an algorithm for the SSRP problem, we first build a $\text{SDO}(1)$. Then, for each $t \in G$ and each $e \in st$ path, we call $\text{QUERY}(s, t, e)$ and output the answer. Thus, we claim the following lemma:

▶ **Lemma 7.** *If we can build a $\text{SDO}(1)$ with query time q in time T, then there is an algorithm for SSRP problem with a running time $O(T + q|\mathcal{R}|)$, where $|\mathcal{R}|$ is the output size in G.*

The above lemma, along with Theorem 2 implies Theorem 6.

1.2 Related Work

Other related problems include the fault-tolerant subgraph problem. In this problem, we want to find a subgraph of G such that the shortest path from s is preserved in the subgraph after any edge deletion. Parter and Peleg [26] designed an algorithm to compute a single fault-tolerant subgraph with $O(n^{3/2})$ edges. They also showed that their result could be easily extended to multiple sources. This result was later extended to dual fault by Parter [16] with $O(n^{5/3})$ edges. Gupta and Khan [19] extended the above result to multiple sources. All the above results are optimal due to a result by Parter [25] which states that a multiple source f-fault tolerant subgraph requires $\Omega\left(n^{2-\frac{1}{f+1}}\right)$ edges. Bodwin et al. [7] showed the existence of a f-fault tolerant subgraph of size $O\left(fn^{2-\frac{1}{2^f}}\right)$.

2 Preliminaries

Let $G(V, E)$ be an undirected unweighted graph with a source s. Given two vertices u and v in a graph H, unless otherwise stated, $(uv)_H$ denotes the shortest path from u to v in H. If $H = G$, we will remove the subscript and the brackets – we will apply this policy for all the

---

[2] In a RAM model with words of $O(\log n)$ bits.

notations below. $|uv|_H$ denotes the length of the shortest path in H. Some of our graphs will be weighted, even though G is unweighted. If H is weighted, then we will abuse notation and use $|uv|_H$ to denote the weight of the shortest path from u to v in H. The edges and vertices of H will be denoted by E_H and V_H, respectively. Additionally, m_H and n_H will denote the number of edges and vertices in H, respectively. $\text{SPT}_H(s)$ denotes the shortest path tree from s in H. We can view the $\text{SPT}_H(s)$ to be drawn from top to bottom with the top vertex being s. For any two vertices u, v on the st path in $\text{SPT}_H(s)$, we say that u is before / above v if $|su|_H < |sv|_H$. Similarly, we say that u is after/below v if $|su|_H > |sv|_H$. For an edge e in a weighted graph H, $wt_H(e)$ will denote the weight of e. Given two paths $(uv)_H$ and $(vw)_H$, the path $(uv)_H + (vw)_H$ denotes their concatenation. $P[u, v]$ denotes a contiguous subpath of P starting at u and ending at v. Sometimes, we may also write *the interval $[u, v]$ of P* to denote $P[u, v]$. We say u *comes before* v on a path R starting from s, if $|R[s, u]| < |R[s, v]|$. Similarly, we can define the term u *comes after* v on path R.

A replacement path R is the shortest path from s to t avoiding an edge e on st path. There can be many replacement paths of the same length avoiding e. To ensure uniqueness, we will use the following definition of replacement path[3].
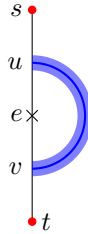
■ **Figure 1** Replacement path with detour uv and detour point u.

▶ **Definition 8** (Replacement Path)**.** *A path R from s to t avoiding e is called a replacement path if (1) it diverges from and merges to the st path just once (2) its divergence point from the st path is as close to s as possible. (3) it is the* **lexicographically smallest**[4] *shortest path in G satisfying (1) and (2).*

We now define some terms related to replacement paths. $(st \diamond e)_H$ denotes the replacement path from s to t avoiding the edge e in H. We can generalize this notation to a replacement path that avoids a set of edges. Thus, $(st \diamond F)_H$ denotes the replacement path from s to t in H avoiding a set F of edges. In our algorithm, after we find the replacement path $(st \diamond e)_H$, we will store its length in $d_H(s, t, e)$. Sometimes, we also want to store the length of $(st \diamond F)_H$. In that case, we will store it in $d_H(s, t, F)$.

▶ **Definition 9** (Detour and Detour point of a replacement path)**.** *Let $R = st \diamond e$. Then, the detour of R is $R \setminus st$. That is, let us assume that R leaves st above e at a vertex u, and merges back on st at vertex v after e, then detour of R is $R[u, v]$. Also, the vertex at which the detour starts is called the detour point of R. So, u is the detour point of R or in short $\text{DP}(R) = u$.*

---

[3] This was referred to as *preferred replacement path* in [19].

[4] Let P and P' first diverge from each other to $x \in P$ and $x' \in P'$ respectively. If the index of x is lower than x', then P is said to be lexicographically smaller than P'.

Lastly, in our algorithm, we will need to find least common ancestor of any two vertices u and v in $\text{SPT}_H(s)$. Let $\text{LCA}_H(u, v)$ denotes the least common ancestor of u and v in $\text{SPT}_H(s)$. To find the LCA, we will use the following result:

▶ **Lemma 10** (See [1] and its references). *Given a tree T on n vertices, we can build a data structure of size $O(n)$ in $O(n)$ time such that the least common ancestor query can be answered in $O(1)$ time.*

3 Overview of our algorithm to build SDO(1)

We will use divide and conquer approach to build SDO(1). This strategy has been previously used for directed graphs in [16, 11]. However, simply using this strategy will not get us close to our desired bound of $\widetilde{O}(m\sqrt{n})$. For that, we need to combine this divide and conquer strategy with an idea of Gupta and Singh [20]. This combination is one of the technical contributions of the paper.

Like [16, 11], we use the following separator lemma to divide the graph G.

▶ **Theorem 11** (Separator Lemma [16, 11]). *Given a tree T with n nodes rooted at a source s, one can find in $O(n)$ time a vertex r that separates the tree T into edge disjoint sub-trees M, N such that $E_M \cup E_N = E_T, V_M \cap V_N = \{r\}$ and $\frac{n}{3} \leq |V_M|, |V_N| \leq \frac{2n}{3}$.*

Without loss of generality, we will assume that $s \in V_M$. Thus, r is the root of N. Also, note that s and r may or may not be the same. Let G_M and G_N be the graph induced by the vertices of M and N, respectively. There is one more important term that we will use in our paper:

▶ **Definition 12** (Primary Path $\mathcal{P}$). *Using the separator lemma, $\text{SPT}(s)$ can be divided into two sub-trees M and N with roots s and r. The path from s to r is called the primary path and is denoted by $\mathcal{P}$.*

We now describe our data structure that we will build recursively. We can view the data structure as a binary tree $\mathcal{T}$. The root contains data structure for the entire graph G. We will abuse notation and say the root is the graph G.

The left child of the root will contain the graph G_M and some weighted edges – we will describe the utility of these weighted edges in the next section. These weighted edges are not in G but are added by our algorithm. We will then build a data structure for G_M recursively. The right child of the root will contain the graph G_N, again with some weighted edges.

At the root of $\mathcal{T}$, we store the following data structures. For each $v \in G$, set $d(s, v) = |sv|$. Similarly, set $d(r, v) = |rv|$. For each $v \in G_N$, set $d(s, v, G_N) = |sv \diamond G_N|$. Similarly, for each $v \in G_M$, set $d(s, v, G_M) = |sv \diamond G_M|$. All these quantities can be computed using a single source shortest path algorithm in $\widetilde{O}(m + n)$ time. Additionally, we will find the length of all replacement paths from s to r avoiding edges on the primary path $\mathcal{P}$. This can be done in $\widetilde{O}(m + n)$ time using[5] [21]. We will set $d(s, r, e) = |sr \diamond e|$ for each edge $e \in \mathcal{P}$.

We store the above data-structure in each node of $\mathcal{T}$. If a node of $\mathcal{T}$ contains graph H, then we can contruct the above data-structures in $\widetilde{O}(m_H + n_H)$ time. We now describe our algorithm that finds replacements paths using $\mathcal{T}$.

Let us see how we find and store lengths of the replacement paths at the root of $\mathcal{T}$, that contains graph G. First, we find the replacement paths for edges on the primary path. Let $R = st \diamond e$ where $e \in \mathcal{P}$. We define R to be either *jumping* or *departing* depending on whether it merges back to the primary path or not.

---

[5] This algorithm work for graphs with non-negative edge weights. And our graph may have weighted edges.

▶ **Definition 13** (Jumping and Departing paths [11, 17])**.** *Let $R = st \diamond e$ where $e \in \mathcal{P}$. R is called a* **jumping path** *if it uses some vertex $u \in \mathcal{P}$ after e. If the path is not jumping then it is a* **departing path***. If a replacement path is jumping, then it is called* jumping replacement path*. Similarly, we define departing replacement path. See Figure 2 for a visualization of these two kinds of paths.*

(a) Departing Replacement Path.   **(b)** Jumping Replacement Path.

🟨 **Figure 2** Departing and Jumping Replacement Path.

Note that, jumping or departing path is defined only when the edge fault is on the primary path. Also, if a replacement path is departing, then the destination t cannot lie on $\mathcal{P}$. In Section 4, we will find all jumping replacement paths.

In Section 5, we design a new algorithm for finding and storing all departing replacement paths. To this end, we will use the main idea in the paper of Gupta and Singh [20]. In [20], the authors sampled a set of vertices with probability of $O(\frac{1}{\sqrt{n}})$. Then, for a vertex $t \in G$, they find a sampled vertex near t on the st path. They call this vertex t_s. Then, they show the following important lemma, which is the main idea of their paper:

▶ **Lemma 14** (Lemma 11 in [20])**.** *The number of replacement paths from s to t that avoid edges in st_s path and also avoid t_s is $O(\sqrt{n})$.*

An astute reader can see that the definition of replacement paths in the above definition looks very similar to departing replacement paths. We prove that this is indeed the case. Thus, we can transfer the result in Lemma 14 to departing replacement paths. This is the main novelty of the paper. The main technical result of Section 5 is as follows:

▶ **Lemma 15.** *For each $t \in G$, all departing replacement paths to t can be found in deterministic $\widetilde{O}(m\sqrt{n})$ time. Moreover, the length of all such departing paths can be stored in a data structure of size $\widetilde{O}(n\sqrt{n})$ and can be queried in $\widetilde{O}(1)$ time.*

4   Algorithm to build SDO(1): Replacement paths that are not departing

In Section 5, we will find all and store all departing replacement paths. Thus, we just need to concentrate on the replacement paths that are either jumping or the faulty edge $e \notin \mathcal{P}$. We now divide remaining replacement paths depending on where the destination t and faulty edge e lies. There are following cases:

(a) $e \in \mathcal{P}$.　　**(b)** $e \notin \mathcal{P}$.

■ **Figure 3** $e \in G_M$ and $t \in G_N$.

4.1　$e \in G_M$ and $t \in G_N$

This case itself can be divided into two cases depending on whether e lies on the primary path or not.

1. $e \in \mathcal{P}$ (See Figure 3(a))

   Let $R = st \diamond e$. If R is departing then we will see how to find it in Section 5. So, assume that R is jumping . This implies that R merges back to $\mathcal{P}$ at a vertex, say w, after the edge e. Since $t \in G_N$, $st = sw + wr + rt$. Thus, after merging with $\mathcal{P}$ at w, the replacement path passes through r. In that case, $|st \diamond e| = |sr \diamond e| + |rt|$. We can easily find the right hand side of the above equality as we have stored $d(s, r, e) = |sr \diamond e|$ and $d(r, t) = |rt|$.

2. $e \notin \mathcal{P}$ (See Figure 3(b))

   In this case, we claim that $st \diamond e = st$. The st path has $\mathcal{P}$ as its prefix. Since $\mathcal{P}$ lies in G_M and survives after the deletion of e, st path remains intact.

4.2　$e \in G_M$ and $t \in G_M$

Since both e and t lie in G_M, one may think that we can recurse our algorithm in G_M to find $st \diamond e$. If $st \diamond e$ completely lies inside G_M, this is indeed the case. However, $st \diamond e$ may also use edges of G_N. To handle such cases, before recursing in G_M, we will add weighted edges to it. For each $v \in G_M$, we will add an edge from r to v with a weight $|rv \diamond G_M|$. We have already calculated this weight, it is stored in $d(r, v, G_M)$. Let the set of weighted edges added to G_M be called X. We now look at two cases, (1) $e \in \mathcal{P}$ and (2) $e \notin \mathcal{P}$.
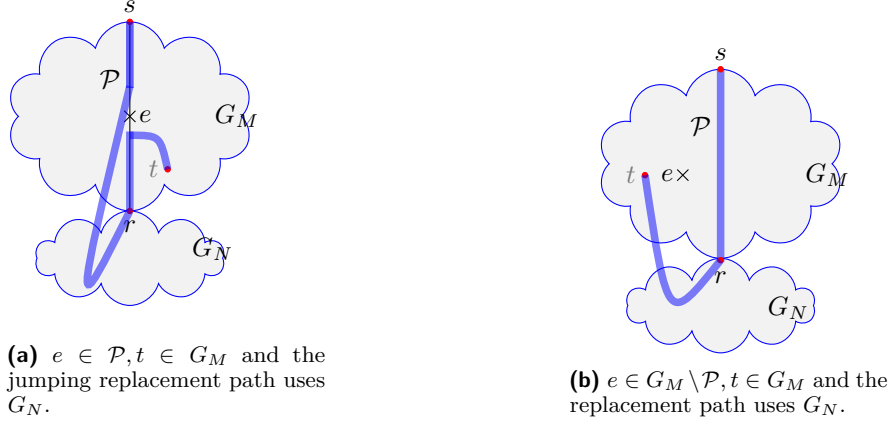
4.2.1　$e \in \mathcal{P}$

Let $R = st \diamond e$ be a jumping replacement path. We will show that $st \diamond e = sr \diamond e + rt$. As we have calculated the length of both the paths in the right-hand side of the above equality, there is no need to even recurse in this case. To prove the above equality, we first prove the following simple lemma:

▶ **Lemma 16.** *Let $e \in \mathcal{P}$, $t \in G_M$. Assume that the jumping replacement path $R = st \diamond e$ uses some edges of G_N. Then $st \diamond e$ passes through r.*

Proof. Since R is jumping, it merges with $\mathcal{P}$. There are two ways in which R can merge with $\mathcal{P}$.

1. R merges with $\mathcal{P}$ and then visits the edges of G_N.

   Let us assume that u is the last vertex of G_N in the path R and R merges with $\mathcal{P}$ at w. Since R first merges with $\mathcal{P}$ and then visits the edges of G_N, u comes after w on R.

(a) $e \in \mathcal{P}, t \in G_M$ and the jumping replacement path uses G_N.

(b) $e \in G_M \setminus \mathcal{P}, t \in G_M$ and the replacement path uses G_N.

■ **Figure 4** $e \in G_M$ and $t \in G_M$.

Since w is below e on $\mathcal{P}$, we claim that the sub-path wu of su survives in $G \setminus e$ and is also the shortest path from w to u. But wu path passes through r. Thus, $st \diamond e$ passes through r by construction in Theorem 11.

2. R visits an edge of G_N and then merges with $\mathcal{P}$ (See Figure 4(a))
    In this case, we will show that R merges with $\mathcal{P}$ at r. For contradiction, let w be the vertex at which R merges with $\mathcal{P}$ such that $w \neq r$. Let u be the first vertex of G_N visited by R. Also, w lies after u on path R. Thus, the replacement path $R = R[s, u] + R[u, w] + R[w, t]$. Since w lies below e on $\mathcal{P}$, wu sub-path of su does not contain e and is also the shortest path from w to u. Thus, $R[u, w] = uw$. But uw path passes through r. This implies that R merges with $\mathcal{P}$ at r contradicting our assumption that $w \neq r$. ◀

We are now ready to prove the main lemma in this subsection.

▶ **Lemma 17.** *Let $e \in \mathcal{P}$ and $t \in G_M$. Assume that the jumping replacement path $R = st \diamond e$ uses some edges of G_N. Then $|st \diamond e| = |sr \diamond e| + |rt|$.*

Proof. Using Lemma 16, R passes through r. So, we have $R = R[s, r] + R[r, t]$. The first summand on the right hand side of the above equality represent a path from s to r avoiding e. Thus, $|R[s, r]| = |sr \diamond e|$.

We will now show that $|R[r, t]| = |rt|$. Clearly $|R[r, t]| \geq |rt|$ as the first path avoid e and the second path may or may not. We will now show that the second path also avoids e which will imply that both paths are of same length. For contradiction, assume that rt passes through e. Let us assume that there is a vertex w before the edge e on path $\mathcal{P}$ such that $rt = rw + wt$. Thus, rw passes through e but wt avoids e. But then, there is a path R' such that $R' = sw + wt$ which avoids e. We claim that $|R'| < |R|$ contradicting the fact that R is the replacement path from s to t avoiding e.

To this end,

$$
\begin{aligned}
|R| &= |sr \diamond e| + |R[r, t]| \\
&\geq |sr \diamond e| + |rt| \\
&= |sr \diamond e| + |rw| + |wt| \\
&\geq |sr \diamond e| + |wt|
\end{aligned}
$$

Since $|sw| < |sr|$, $|sw| < |sr \diamond e|$

$$
\begin{aligned}
&> |sw| + |wt| \\
&= |R'|.
\end{aligned}
$$

This completes the proof of the lemma. ◀

4.2.2 $e \notin \mathcal{P}$

In this case, we will show a path in $G_M \cup X$ such that it avoids e and has the same length as $st \diamond e$. Please see Figure 4(b) for a visualization of this case.

▶ **Lemma 18.** *Let $e \in G_M \setminus \mathcal{P}$ and $t \in G_M$. Assume that the replacement path $R = st \diamond e$ uses some edges of G_N. Then there is a path in $G_M \cup X$ that avoids e and has length $|st \diamond e|$.*

Proof. Let us first prove that R can alternate between edges of G_N and G_M just once. To this end, let u be the last vertex of G_N visited by R. Thus, $R = R[s, u] + R[u, t]$. But the shortest path su remains intact in $G \setminus e$ as $e \notin \mathcal{P}$ and the shortest su path passes through r. This implies that $R = R[s, u] + R[u, t] = R[s, r] + R[r, u] + R[u, t]$. By construction, the first path $R[s, r] = sr$ and it completely lies in G_M. The second path $R[r, u] = ru$ and it completely lies in G_N. Let v be the vertex just after u in R. So, $v \in G_M$. So we have $R = sr + ru + (u, v) + R[v, t]$. By construction, $R[v, t]$ completely lies in G_M. Thus, R alternates from edges of G_N to G_M just once.

From the above discussion $R = R[s, r] + R[r, u] + (u, v) + R[v, t] = R[s, r] + R[r, v] + R[v, t]$. The first and the last paths of the above equality completely lies in G_M. By construction, $R[r, v]$ does not contain any edge of G_M. Thus, $R[r, v]$ is the shortest path from r to v avoiding edges of G_M, that is $|R[r, v]| = |rv \diamond G_M| = d(r, v, G_M)$.

Since we have added an edge from r to v with weight $d(r, v, G_M)$, we will now show that there is a path in $G_M \cup X$ that avoids e and has same weight as R. Consider the path $R' = R[s, r] + (r, v) + R[v, t]$. The reader can check all the three subpaths lie completely in $G_M \cup X$. Moreover, $(r, v) \in X$ is a weighted edge. Thus the weight of $|R'| = |R[s, r]| + wt_{(G_M \cup X)}(r, v) + |R[v, t]| = |R[s, r]| + d(r, v, G_M) + |R[v, t]| = |R[s, r]| + |R[r, v]| + |R[v, t]| = |R|$. This completes the proof. ◀

4.3 $e \in G_N$ and $t \in G_M$

In this case, st path completely lies in G_M and thus survives. Thus, $|st \diamond e| = |st| = d(s, t)$.

4.4 $e \in G_N$ and $t \in G_N$

In this case, we will recurse in G_N. However, G_N may not contain the source s if $s \neq r$. In that case, before recursing, we add a new source s_N in G_N. We also add some *weighted* edges to G_N. For each $v \in G_N$, we add an edge from s_N to v with a weight $d(s, v, G_N) = |sv \diamond G_N|$. Let this set of edges be called Y. Let us now show that we will find all the replacement paths if we recurse in $G_N \cup Y$.

▶ **Lemma 19.** *Let $e \in G_N$ and $t \in G_N$. There is a path from s_N to t in $G_N \cup Y$ that avoids e and has weight $|st \diamond e|$.*

Proof. Let $R = st \diamond e$. Let us first prove that once R visits an edge of G_N, it cannot visit an edge of G_M anymore. Let u be the last vertex of G_M visited by R and v be the vertex after u in R. Thus, $v \in G_N$. Thus, $R = R[s, u] + (u, v) + R[v, t]$. By construction, $R[v, t]$ completely lies in G_N. The shortest path su remains intact in $G \setminus e$ as $e \in G_N$ and the path su completely lies in G_M. Thus, $R[s, u] = su$. So, the path R first visits only edges of G_M(in $R[s, u]$), then goes to G_N (by taking the edge (u, v)) and then remains in G_N (in $R[v, t]$). Thus, R does not visit any edge of G_M once it visits an edge of G_N.

By the above discussion, $R = R[s, u] + (u, v) + R[v, t] = R[s, v] + R[v, t]$ such that $R[v, t]$ completely lies in G_N. Also, $R[s, v]$ completely lies in G_M except the last edge which has one endpoint in G_M and other endpoint $v \in G_N$. Thus, $R[s, v] = |sv \diamond G_N| = d(s, v, G_N)$ and we have $R = sv \diamond G_N + R[v, t]$.

Now consider a path R' that avoids e from s_N to t in $G_N \cup Y$. We construct this path as follows: we will first take the weighted edge $s_N \to v$ and then the path $R[v,t]$. The reader can check that these two subpaths completely lie in $G_N \cup Y$. The weight of this path is $|R'| = |sv \diamond G_N| + |R[v,t]| = d(s,v,G_N) + |R[v,t]| = |R[s,v]| + |R[v,t]| = |R|$. ◀

4.5   One endpoint of e is in G_M and the other is in G_N and $t \in G$

This is an easy case as the st path survives in $G \setminus e$ as st does not contain e. Thus, $st \diamond e = st$

5   Departing Replacement paths

In the previous section, we have already found out a replacement path if it is jumping or if the edge failure is not on the primary path. In this section, we will try to find the best departing path after an edge failure. To this end, we define the following:

▶ **Definition 20** (Candidate departing paths). *Let $e \in \mathcal{P}$ and let D be the set of all path avoiding e that do not use any vertex of $\mathcal{P}$ after e. $P \in D$ is called the candidate departing path for e, if among paths in D, P has the minimum length. In case there are many departing paths avoiding same edge with same length, we will break ties using Definition 8.*

Note that, a candidate departing path may or may not be a replacement path. In case it is, we call it a departing replacement path. Also, P may be a candidate departing path for all edges in an interval, say $yz \in \mathcal{P}$, but may be a replacement path for a sub-interval of yz. With this definition in hand, we will now find all candidate departing paths.

5.1   Finding all candidate departing paths

In the previous section, we added some weighted edges in the graph when we recurse. Thus, there might be two types of edges in the graph – *weighted* and *unweighted*. The weighted edges represent paths in the graph G, and the unweighted edges are present in G. G contains only unweighted edges. However, the graph at an internal node of $\mathcal{T}$, say graph $\widehat{G}$, may have weighted as well as unweighted edges.

In the ensuing discussion, let $\widehat{G}$ be the graph processed by our algorithm at some internal node of $\mathcal{T}$. In the graph $\widehat{G}$, we will assume that there is a source s. Let $\overline{G}$ be the parent of $\widehat{G}$. In our algorithm, we partition $\overline{G}$ into two disjoint graphs and then recurse on it. If $\widehat{G}$ is the left child of $\overline{G}$, then it includes a set X of weighted edges added by us in Section 4.2. Similarly, if $\widehat{G}$ is the right child of $\overline{G}$ then it includes a set Y of weighted edges added by us in Section 4.4. Using the separator lemma, we will find the vertex r that partitions $\mathrm{SPT}_{\widehat{G}}(s)$. Also, the primary path $\mathcal{P} = sr$.

We now give an overview of our method to find candidate departing paths in $\widehat{G}$. To this end, we will use the main idea in the paper of Gupta and Singh [20]. In [20], the authors proved Lemma 14. Though they did not mention it, the paths in the Lemma 14 look very much like the departing paths. Indeed, that lemma is more general than what the authors originally intended it to be. The authors show the above lemma for a specific vertex t_s, but a careful reading of the paper suggests that the above lemma is true for any vertex on the st path. We now generalise this lemma. However, there is one problem. The above result holds only for an unweighted graph, whereas $\widehat{G}$ is weighted. Thus, we cannot prove the above lemma as it is. However, we will prove the following weaker lemma:

▶ **Lemma 21.** *Let $\widehat{G}$ be the graph at an internal node in the binary tree $\mathcal{T}$. Let s be the source of $\widehat{G}$. For a destination $t \in \widehat{G}$, let p be any vertex on st path. The number of replacement paths from s to t that avoid edges on sp path and also avoid p is $O(\sqrt{n})$.*

Some discussion is in order. In an ideal case, the number of replacement paths that avoid edges on sp and also avoid p should have been $O(\sqrt{n_{\widehat{G}}})$. This result would have been similar to Lemma 14. Unfortunately, we cannot prove this result as $\widehat{G}$ is weighted. However, if we just expand the weighted edge in the graph $\widehat{G}$, then we will get an unweighted graph. By expanding, we mean that for each weighted edge, add the path that represents that weighted edge. However, this process may increase the number of vertices in the graph to n. Now, we can adapt Lemma 14. For an unweighted graph with n vertices, this lemma guarantees that the number of replacement paths avoiding p will be at most $\sqrt{n}$. Indeed, this is our result in Lemma 21. Interested reader can find the proof of this in the full version of the paper [13] as it is an extension of the proof in [20].

Lemma 21 implies that there are only $O(\sqrt{n})$ replacement paths that have some special properties which are similar to the properties of departing paths. So, our plan of action is as follows:

1. Show that there are only $O(\sqrt{n})$ candidate departing paths to t in $\widehat{G}$. This will be done by showing the similarity between the replacement paths in Lemma 21 and candidate departing paths.

2. Show that we can find the lengths of all the candidate departing paths in $\widehat{G}$ in $O(m_{\widehat{G}}\sqrt{n})$ time. Additionally, we show that we can store the lengths in a compact data structure of size $O(n_{\widehat{G}}\sqrt{n})$. Given any query $\text{QUERY}(s, t, e)$ to this data-structure such that $s, t \in \widehat{G}$ and e is on the primary path of $\widehat{G}$, we can find the length of corresponding candidate departing path in $\widetilde{O}(1)$ time.

This completes the overview of our algorithm for finding candidate departing paths.

5.2 Similarity between candidate departing paths and replacement paths in Lemma 21

Let us first prove some simple results that will help us prove this section's main idea.

▶ **Lemma 22.** *Let $t \in \widehat{G}$ and $p = \text{LCA}(t, r)$. All the candidate departing paths from s to t avoiding edges on sp path also avoid p. For any edge e on pr path , $st \diamond e = st$.*

Proof. Let $R = st \diamond e$ where $e \in sp$ and R is departing. The detour of R must start above e on $\mathcal{P}$. Since R is departing, it can not merge with the path $\mathcal{P}$ again. So, it avoids p also.

The st path passes through p and does not use any vertex of $\mathcal{P}$ below p. So, removing any edge on pr path does not disturb st path. So, for any edge $e \in pr$, $st \diamond e = st$. ◀

We now show that the candidate departing paths and the replacement paths in Lemma 21 have the same property.

▶ **Lemma 23.** *For any vertex $t \in \widehat{G}$, there are $O(\sqrt{n})$ candidate departing paths to t.*

Proof. The proof of the lemma can be derived using Lemma 22 and Lemma 21 together. For complete proof please see the full version [13]. ◀

Given the above lemma, we need to store $O(\sqrt{n})$ candidate departing paths to t in $\widehat{G}$. Before designing an algorithm to find candidate departing paths, we first see how we plan to store these paths in a compact data structure.

5.3 The data-structure at each node of $\mathcal{T}$

Let us first discuss a small technical detail that may be perceived as a problem but is not. Our graph $\widehat{G}$ is weighted. It stands to reason that even the primary path $\mathcal{P}$ in $\widehat{G}$ maybe weighted. Since candidate departing paths are only for the faults on the primary path, it may not be clear what happens if the edge on the primary path is weighted. To this end, we show that on any st path in $\widehat{G}$, all edges except may be the first one, are unweighted.

▶ **Lemma 24.** *For $t \in \widehat{G}$, except may be the first edge of $(st)_{\widehat{G}}$, all other edges of $(st)_{\widehat{G}}$ are unweighted.*

Proof. We will prove using induction on the nodes of $\mathcal{T}$. In the root of $\mathcal{T}$, we have the graph G. Clearly, the graph G satisfies the property of the lemma. Let us assume using induction that the parent of the graph $\widehat{G}$ satisfies the property of the lemma. Let $\overline{G}$ be the parent of $\widehat{G}$. Let $\overline{s}$ be the source in $\overline{G}$. Thus, in the graph $\overline{G}$, using the separator lemma, we find a $\overline{r}$ that divides $\overline{G}$ into two parts. The path from $\overline{s}$ to $\overline{r}$, say $\overline{\mathcal{P}}$ is the primary path. By induction, only the first edge of the primary path may be weighted. There are two cases:
1. $\widehat{G}$ is the left child of $\overline{G}$.
   In this case, the source of $\widehat{G}$ is also $\overline{s}$. For each $t \in \widehat{G}$, it can be observed that the path $(\overline{s}t)_{\overline{G}} = (\overline{s}t)_{\widehat{G}}$. Using induction hypothesis, since $(\overline{s}t)_{\overline{G}}$ satisfies the statement of lemma, so does $(\overline{s}t)_{\widehat{G}}$.
2. $\widehat{G}$ is the right child of $\overline{G}$.
   There are two cases. When $\overline{s} = \overline{r}$, then we fall back in point(1). So, let us look at the case when $\overline{s} \neq \overline{r}$.
   For a $t \in \widehat{G}$, by construction, $(\overline{s}t)_{\overline{G}}$ passes through $\overline{r}$. Thus, $(\overline{s}t)_{\overline{G}} = (\overline{s}\overline{r})_{\overline{G}} + (\overline{r}t)_{\overline{G}}$. In $\widehat{G}$, we add a new source s. Also, we add a weighted edge from s to $\overline{r}$ in $\widehat{G}$. The weight of this edge is $|\overline{s}\overline{r} \diamond \widehat{G}|_{\overline{G}}$. Also, $(\overline{r}t)_{\overline{G}} = (\overline{r}t)_{\widehat{G}}$. This implies that there is path in $\widehat{G}$ from s to t, $(s, \overline{r}) + (\overline{r}t)_{\widehat{G}}$. By induction, we claim that on this path except $(s, \overline{r})$, all the edges are unweighted. ◀

Using the above lemma, all except the first edge of the primary path are unweighted. The weighted edges represent edges for which we have already found candidate departing paths at the parent or an ancestor of $\widehat{G}$. Thus, we will only find candidate departing paths for unweighted edges in $\mathcal{P}$. In the ensuing discussion, whenever we mention a path avoiding an edge on the primary path, it will always refer to an unweighted edge.

We now prove some simple lemmas that will help us build data structures for candidate departing paths.

▶ **Lemma 25.** *Let R and R' be two different candidate departing paths from s to t avoiding edges e and e' respectively on the path $\mathcal{P}$. If e lies above e' on $\mathcal{P}$, then $|R| > |R'|$.*

Proof. The detour of the candidate departing R starts before e, and once it departs, it does not merge with $\mathcal{P}$ again. As e lies above e' on $\mathcal{P}$, R also avoids e'. If $|R| \leq |R'|$, then by Definition 8, R must be the candidate departing path avoiding e', leading to a contradiction. So, it must be the case that $|R| > |R'|$. ◀

▶ **Lemma 26.** *Let R be a candidate departing path. Let yz be the maximal subpath of $\mathcal{P}$ such that R is the candidate departing path for edges in yz. Then $\text{DP}(R) = y$.*

Proof. For complete proof of the lemma please see the full version [13]. ◀

The above lemma states that if R avoids edges in yz, then the detour of R necessarily starts from y. We will now prove the contrapositive of the Lemma 25.

▶ **Lemma 27.** *Let R and R' be two different candidate departing paths from s to t. If $|R| > |R'|$, then $\mathrm{DP}(R)$ lies above $\mathrm{DP}(R')$ on $\mathcal{P}$.*

Proof. For complete proof of the lemma please see the full version [13]. ◀

We will now use the above lemmas and our deduction to build a compact data-structure for all candidate departing paths. To this end, we will store an array $\mathrm{DEP}(t)$ for each $t \in \widehat{G}$. $\mathrm{DEP}(t)$ will store candidate departing paths from s to t in increasing order of their lengths. By Lemma 23, there are $O(\sqrt{n})$ such paths. let us denote them by $R_1, R_2, \ldots, R_k$ where $k = O(\sqrt{n})$. For any two consecutive candidate departing paths R_i and R_{i+1}, using Lemma 27 and Lemma 26, we claim that R_{i+1} is the candidate departing path avoiding edges in $[\mathrm{DP}(R_{i+1}), \mathrm{DP}(R_i)]$ on the primary path $\mathcal{P}$. Since the size of $\mathrm{DEP}(t) = O(\sqrt{n})$, the total size of DEP data-structure is bounded by $O(n_{\widehat{G}}\sqrt{n})$.

▶ **Lemma 28.** $\sum_{t \in \widehat{G}} size\ of\ \mathrm{DEP}(t) = O(n_{\widehat{G}}\sqrt{n})$

5.4 Finding and storing all candidate departing paths efficiently

Let us first describe the setting that will be used throughout this section. At an internal node $\widehat{G}$ of $\mathcal{T}$, we are planning to find all candidate departing paths from the source s. To this end, we will find a vertex r that will divide $\mathrm{SPT}_{\widehat{G}}(s)$ roughly equally. Also, $\mathcal{P} = sr$.

To find all candidate departing path, we will build an auxiliary graph G which we will build incrementally. The source in this graph is (s). All other vertices in G are tuples of the form $(v, |R|)$, where $v \in \widehat{G} \setminus \mathcal{P}$ and R is a candidate departing path to v. During initialization, we will add $(v, |sv|_{\widehat{G}})$ in G for each $v \in \widehat{G} \setminus \mathcal{P}$. Also, there will be an edge from (s) to $(v, |sv|_{\widehat{G}})$ with weight $|sv|_{\widehat{G}}$. We will show the following property at the end of our analysis.

▶ **Property 29.** *Let R be the candidate departing path to v avoiding edge on the subpath yz of $\mathcal{P}$. Then, there is a vertex $(v, |R|)$ in G. Moreover the shortest path from (s) to $(v, |R|)$ in the graph G is of length $|R|$, that is $|R| = |(s)(v, |R|)|_G$*

In Lemma 30, we will show that Property 29 is true for all the nodes added during initialization. Also, we will create $\mathrm{DEP}(v)$ for each v during intialization. We will store candidate departing paths in $\mathrm{DEP}(v)$ in increasing order of lengths. Given a departing path R, we will assume that we will store the following information about R in $\mathrm{DEP}()$.

1. The endpoints of R.
2. The weight of path R.
3. The last edge of R and its weight.
4. $\mathrm{DP}(R)$.

After initialization, we will run a variant of Dijkstra's algorithm in G. To this end, we will construct a min-heap H in which we will store all the departing paths that we have discovered at any point in the algorithm. We use the first two points of Definition 9 to select the minimum element from H,i.e., given two candidate departing paths R and R', R is *smaller* than R' if $|R| < |R'|$ or $|R| = |R'|$ and $\mathrm{DEP}(R)$ is closer to s than $\mathrm{DEP}(R')$. If $\mathrm{DEP}(R) = \mathrm{DEP}(R')$, then we can break ties arbitrarily.

We now explain the adaptation of Dijkstra's algorithm. After initialization,for each $(v, |sv|)$, and for each neighbor neighbor w of v, we add the departing path $sv + (v, w)$ in H if $w \notin \mathcal{P}$. Then, we go over all the elements of the heap till it is empty. Let us assume that R is the minimum element of the heap and it ends at v and $(u, v) \in \widehat{G}$ is the last edge of R. This implies that R was added in H while processing a candidate departing path for u. Let this path be R_u. Thus, there is a node $(u, |R_u|)$ in G. We now need to decide whether R is a candidate departing path for v.

To this end, we look at the last candidate departing path added by us in $\text{Dep}(v)$, let it be Q. We then check if the $d_{\widehat{G}}(s, \text{Dp}(R))$ is less than $d_{\widehat{G}}(s, \text{Dp}(Q))$. If yes, then we have found a new candidate departing path to v that avoids all edges in $[\text{Dp}(R), \text{Dp}(Q)]$ of $\mathcal{P}$. Thus, we will add the vertex $(v, |R|)$ in the graph with an edge of weight $wt_{\widehat{G}}(u, v)$ from $(u, |R_u|)$. Also, for each neighbor w of v, we will add the departing path $R + (v, w)$ to the heap if $w \notin \mathcal{P}$. For the formal pseudocode, see the full version [13] of this paper.

5.5 Correctness and running time of the algorithm storing candidate departing paths

We claim that, the time taken to construct the $\text{Dep}()$ data-structure at a node of the binary tree with graph $\widehat{G}$ is $O(m_{\widehat{G}} \sqrt{n})$. Moreover, the size of the data-structure is $O(n_{\widehat{G}} \sqrt{n})$. At first, we prove that our algorithm stores correct lengths of all candidate departing paths.

▶ **Lemma 30.** *Let R be a candidate departing path to v where $v \in \widehat{G} \setminus \mathcal{P}$. Let yz be the maximal subpath of $\mathcal{P}$ such that R is the candidate departing path for edges in yz. Then $(v, |R|) \in G$ and satisfies Property 29.*

Proof. We will prove the above lemma using induction on the weighted distance of a vertex from (s) in G. During initialization, for each $v \in G \setminus \mathcal{P}$, we add a vertex $(v, |sv|_{\widehat{G}}) \in G$. Also, the weight of the edge from (s) to $(v, |sv|_{\widehat{G}})$ is $|sv|_{\widehat{G}}$. We claim that the statement of the lemma is true for the smallest candidate departing path to v. Indeed, using Lemma 22, $(sv)_{\widehat{G}}$ is the a replacement path for the edges in subpath pr on $\mathcal{P}$ where $p = \text{Lca}_{\widehat{G}}(v, r)$. Also, $(sv)_{\widehat{G}}$ is the smallest replacement path because it is the shortest path from s to v in $\widehat{G}$. Thus, the base case is true for all $v \in \widehat{G} \setminus \mathcal{P}$.

Let us now assume that the statement of the lemma is true for all candidate departing paths to v with length $< |R|$. Let the second last vertex of R be u. Since R is a candidate departing path, even $R \setminus (u, v)$ is a candidate departing path. Since $R[s, u]$ has length less than R, by induction hypothesis, there is a vertex $(u, |R \setminus (u, v)|)$ in G. Also there is at least one replacement path in $\text{Dep}(v)$ of weight less than $|R|$ – as we have added $(sv)_{\widehat{G}}$ in $\text{Dep}(v)$. Let Q be a candidate departing path of largest weight less than the weight of R. Let us also assume that Q avoids edge on subpath $zz' \in \mathcal{P}$. Thus, using Lemma 26, $\text{Dp}(Q) = z$. Since $|Q| < |R|$, using Lemma 27, $\text{Dp}(R)$ lies above $\text{Dp}(Q)$ on $\mathcal{P}$. Using the induction hypothesis, there is a vertex $(v, |Q|)$ in G.

We will now show that our algorithm will add $(v, |R|)$ in G. There are four cases:
1. Our algorithm does not add any vertex for v after $(v, |Q|)$
   But our algorithm does process the vertex $(u, |R \setminus (u, v)|)$. Thus, it will check each neighbour of u. When it checks the neighbor v, it will add the departing path R in the heap H. Thus, we will add the vertex $(v, |R|)$ in G while processing R, leading to a contradiction.
2. Our algorithm adds a vertex $(v, |R'|)$ where $\text{Dp}(R)$ lies above $\text{Dp}(R')$ on $\mathcal{P}$
   We claim that the weight of R' cannot be less than the weight of R as then R is not the candidate departing path for all the edges in $\text{Dp}(R')z$ subpath, contradicting the statement of the lemma. So let us assume that $|R| = |R'|$. But then $\text{Dp}(R)$ lies above $\text{Dp}(R')$ on $\mathcal{P}$. Thus, the min-heap will give preference to the replacement path R first, and our algorithm will make the vertex $(v, |R|)$. Again a contradiction.
3. Our algorithm adds a vertex $(v, |R'|)$ where $\text{Dp}(R)$ lies below $\text{Dp}(R')$ on $\mathcal{P}$
   Again, we claim that the weight of R' cannot be less than the weight of R as then it R is not the replacement path for all edges in yz subpath, contradicting the statement of the

lemma. So let us assume that $|R| = |R'|$. But $\text{DP}(R')$ is closer to s than $\text{DP}(R)$. Thus, R' should be the candidate departing path avoiding edges of yz. This contradicts our assumption that R is the candidate departing path for all edges in yz.

4. Our algorithm adds a vertex $(v, |R'|)$ where $\text{DP}(R) = \text{DP}(R')$
   $|R'|$ cannot be less than $|R|$ as otherwise our algorithm will give preference to path R. But, if $|R| = |R'|$, then there is a vertex $(v, |R'|) = (v, |R|)$ in G.

So, we add the node $(v, |R|)$ in the graph G. At that moment, we also adds an edge from $(u, R \setminus (u, v))$ to $(v, |R|)$ in G with weight $wt_{\widehat{G}}(u, v)$. Using induction, $|(s)(u, R \setminus (u, v))|_G = |R \setminus (u, v)|$. Thus, $|(s)(v, |R|)|_G = |R \setminus (u, v)| + wt_{\widehat{G}}(u, v) = |R|$. This completes our proof. ◄

Let's determine the running time of our algorithm. Using Lemma 23, for each $v \in \widehat{G}$, we make $O(\sqrt{n})$ entries in $\text{DEP}(v)$. In other words, we make $O(\sqrt{n})$ nodes of type $(v,)$ in G. Whenever we add a node $(v, |R|)$ in G, we see all the edges of v. This implies that the total time taken to process all the vertices of v in G is $O(\sqrt{n} \deg_{\widehat{G}}(v))$. Summing it over all the vertices gives us the bound of $O(m_{\widehat{G}} \sqrt{n})$. Using a similar calculation, the total size of our data-structure for $\widehat{G}$ is $O(n_{\widehat{G}} \sqrt{n})$. Thus, we claim the following lemma:

▶ **Lemma 31.** *The time taken to construct the $\text{DEP}()$ data-structure at a node of the binary tree with graph $\widehat{G}$ is $O(m_{\widehat{G}} \sqrt{n})$. Moreover, the size of the data-structure is $O(n_{\widehat{G}} \sqrt{n})$.*

5.6 Querying for a candidate departing path

In this section, we describe how to find a candidate departing path using our data-structure $\text{DEP}()$. Let $t \in \widehat{G} \setminus \mathcal{P}$ and $e \in \mathcal{P}$ be an edge on st path. Let $st \diamond e$ be a candidate departing path, then we can find it using the algorithm given in the full version of this paper [13].

In this algorithm, we perform a binary search in $\text{DEP}(t)$ to find two consecutive paths R and Q such that e lies in the interval $[\text{DEP}(R), \text{DEP}(Q)]$ of $\mathcal{P}$. Using Lemma 27 and Lemma 26, R is the candidate departing path avoiding e.

6 Construction time, size and query time of the SDO(1)

In this section, we show that the construction time of our algorithm is $\widetilde{O}(m\sqrt{n})$. We also bound the size of the data-structure of our algorithm by $\widetilde{O}(n\sqrt{n})$. We also design a query algorithm with a query time $\widetilde{O}(1)$. This proves the main result of the paper.

At the root of $\mathcal{T}$, except for the recursions, we claim that constructing all other data-structures take $O(m\sqrt{n})$ time. This is beacuse, the construction time is dominated by the time to construct $\text{DEP}()$, which using Lemma 31, is $O(m\sqrt{n})$. At the second level of the tree $\mathcal{T}$, we have two nodes. In the left child, we have the graph $G_M \cup X$. This graph has $m_{G_M} + n_{G_M}$ edges and n_{G_M} vertices. Again applying Lemma 31, the time taken to construct all the data-structures in the left child of root is $(m_{G_M} + n_{G_M})\sqrt{n}$. In the right child of the root, we have the graph $G_N \cup Y$. This graph has $m_{G_N} + n_{G_N}$ edges and $n_{G_N} + 1$ vertices. The $+1$ is for the new root in G_N. Again applying Lemma 31, the time taken to construct all the data-structures in the right child of root is $(m_{G_N} + n_{G_N})\sqrt{n}$. Thus, the total time taken at the second level of $\mathcal{T}$ is $= (m_{G_M} + n_{G_M})\sqrt{n} + (m_{G_N} + n_{G_N})\sqrt{n}$. Since $m_{G_M} + m_{G_N} \leq m$ and $n_{G_M} + n_{G_N} = n + 1$, the total time taken is $\leq (m + n + 1)\sqrt{n}$. Note that $n_{G_M} + n_{G_N} = n + 1$ because r is shared both by G_M and G_N. Since, the number of nodes in $\mathcal{T}$ is $O(n)$, we claim that the number of vertices shared by sibling graphs at any level of $\mathcal{T}$ is $O(n)$. Similar to the second level, we claim that the time taken at level ℓ is $\widetilde{O}((m + n + \#\text{nodes shared at level } \ell)\sqrt{n}) = \widetilde{O}((m + n)\sqrt{n})$. We can assume that our graph

G is connected as we need not even process a component that is not reachable from our source s. Thus, the previous running time bound is equal to $\widetilde{O}(m\sqrt{n})$. Since the height of the tree is $\widetilde{O}(1)$, the total time taken to construct our data-structure is $\widetilde{O}(m\sqrt{n})$. Using the same argument, we can bound the size of the data-structure of our algorithm by $\widetilde{O}(n\sqrt{n})$.

6.1   The Query Algorithm

In this section, we will design our query algorithm that will take s, t, e as its parameter. Additionally, it also takes the root of the tree $\mathcal{T}$ as a parameter which contains data structures of the main graph G. The algorithm then basically goes over all the possible cases described in Section 4 (Please see algorithm 3). Also, the algorithm compares that output with the best candidate departing path given by and return the minimum among them.

■ **Algorithm 1** QUERY$(s, t, e, \mathcal{T})$.

---

1 $mindist \leftarrow \infty$;
   /* Section 4.1   */
2 **if** $e \in G_M, t \in G_N$ **then**
3 | **if** $e \in \mathcal{P}$ **then**
     | | /* if $st \diamond e$ happens to be departing   */
4 | | $mindist \leftarrow$ QUERY-DEP(s, t, e) using DEP$()$ data-structure at $\mathcal{T}$;
     | | /* if $st \diamond e$ happens to be jumping   */
5 | | $mindist \leftarrow \min\{mindist, |sr \diamond e| + |rt|\}$
6 | **else**
7 | | $mindist \leftarrow |st|$

   /* Section 4.2   */
8 **if** $e \in G_M, t \in G_M$ **then**
     | /* Section 4.2.1   */
9 | **if** $e \in \mathcal{P}$ **then**
     | | /* if $st \diamond e$ happens to be departing   */
10 | | $mindist \leftarrow$ QUERY-DEP(s, t, e) using DEP$()$ data-structure at $\mathcal{T}$;
     | | /* if $st \diamond e$ happens to be jumping   */
11 | | $mindist \leftarrow \min(mindist, |sr \diamond e| + |rt|)$

     | /* Section 4.2.2   */
12 | **if** $e \in G_M \setminus \mathcal{P}$ **then**
13 | | $mindist \leftarrow$ QUERY$(s, t, e, \text{left child of } \mathcal{T})$

   /* Section 4.3   */
14 **if** $e \in G_N, t \in G_M$ **then**
15 | $mindist \leftarrow |st|$;

   /* Section 4.4   */
16 **if** $e \in G_N, t \in G_N$, **then**
17 | $mindist \leftarrow$ QUERY$(s, t, e, \text{right child of } \mathcal{T})$;

   /* Section 4.5   */
18 **if** *one endpoint of e is in G_M and other in G_N* **then**
19 | $mindist \leftarrow |st|$;

20 return $mindist$;

The reader can see that the time taken by the algorithm (excluding recursion) is $\widetilde{O}(1)$. Since, at each step in this algorithm, we either go to the left child of a node in the tree $\mathcal{T}$ or to the right child, the number of recursive steps in this algorithm is $\widetilde{O}(1)$ or to be specific $O(\log n)$. Then for each child the call to QUERY-DEP(s, t, e) takes $O(\log n)$ time. This implies that the running time of the query algorithm is $O(\log^2 n)$ or $\widetilde{O}(1)$. Thus, we have proven the main theorem of the paper.

---- **References** ----

1 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.

2 Aaron Bernstein. A nearly optimal algorithm for approximating replacement paths and k shortest simple paths in general graphs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 742–755, USA, 2010. Society for Industrial and Applied Mathematics.

3 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.

4 Aaron Bernstein and David R. Karger. Improved distance sensitivity oracles via random sampling. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 34–43. SIAM, 2008. URL: `http://dl.acm.org/citation.cfm?id=1347082.1347087`.

5 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.

6 Davide Bilò, Luciano Gualà, Stefano Leucci, and Guido Proietti. Multiple-edge-fault-tolerant approximate shortest-path trees. In *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, pages 18:1–18:14, 2016.

7 Greg Bodwin, Fabrizio Grandoni, Merav Parter, and Virginia Vassilevska Williams. Preserving distances in very faulty graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 73:1–73:14, 2017.

8 Shiri Chechik and Sarel Cohen. Near optimal algorithms for the single source replacement paths problem. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2090–2109, 2019.

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 Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. Fault tolerant spanners for general graphs. *SIAM J. Comput.*, 39(7):3403–3423, 2010. `doi:10.1137/090758039`.

11 Shiri Chechik and Ofer Magen. Near optimal algorithm for the directed single source replacement paths problem. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 81:1–81:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

12 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.

13   Dipan Dey and Manoj Gupta. Near optimal algorithm for fault tolerant distance oracle and single source replacement path problem. *CoRR*, abs/2206.15016, 2022. `doi:10.48550/arXiv.2206.15016`.

14   Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 506–515, USA, 2009. Society for Industrial and Applied Mathematics.

15   Ran Duan and Seth Pettie. Approximating maximum weight matching in near-linear time. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, FOCS '10, pages 673–682, Washington, DC, USA, 2010. IEEE Computer Society. `doi:10.1109/FOCS.2010.70`.

16   Fabrizio Grandoni and Virginia Vassilevska Williams. Improved distance sensitivity oracles via fast single-source replacement paths. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 748–757. IEEE, 2012.

17   Fabrizio Grandoni and Virginia Vassilevska Williams. Faster replacement paths and distance sensitivity oracles. *ACM Trans. Algorithms*, 16(1), December 2019. `doi:10.1145/3365835`.

18   Manoj Gupta, Rahul Jain, and Nitiksha Modi. Multiple source replacement path problem. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 339–348. ACM, 2020.

19   Manoj Gupta and Shahbaz Khan. Multiple source dual fault tolerant BFS trees. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 127:1–127:15, 2017.

20   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.

21   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.

22   Kavindra Malik, Ashok K Mittal, and Santosh K Gupta. The k most vital arcs in the shortest path problem. *Operations Research Letters*, 8(4):223–227, 1989.

23   Enrico Nardelli, Guido Proietti, and Peter Widmayer. Finding the most vital node of a shortest path. *Theor. Comput. Sci.*, 296(1):167–177, 2003.

24   Noam Nisan and Amir Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35(1-2):166–196, 2001.

25   Merav Parter. Dual failure resilient BFS structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 481–490, 2015.

26   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.

27   Liam Roditty. On the k shortest simple paths problem in weighted directed graphs. *SIAM J. Comput.*, 39:2363–2376, January 2010. `doi:10.1137/080730950`.

28   Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. In *Proceedings of the 32nd International Conference on Automata, Languages and Programming*, ICALP'05, pages 249–260, Berlin, Heidelberg, 2005. Springer-Verlag. `doi:10.1007/11523468_21`.

29   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`.