Approximate shortest paths avoiding a failed vertex : near optimal data structures for undirected unweighted graphs *

Surender Baswana† Neelesh Khanna‡

Abstract

Let $G = (V, E)$ be an undirected unweighted graph. A path between any two vertices $u, v \in V$ is said to be t-approximate shortest path if its length is at most t times the length of the shortest path between u and v. We address the problem of building a compact data structure which can efficiently answer the following query for any $u, v, x \in V$ and $t > 1$:

Report t-approximate shortest path between u and v when vertex x fails

We present data structures for the single source as well as all-pairs versions of this problem. The query time guaranteed by our data structures is optimal up to a constant factor. Moreover, the size of each of them *nearly* matches the size of the corresponding data structure with no failures.

Keywords: shortest path, distance, approximate distance, oracle.

*The results of the preliminary version of this article appeared in the proceedings of 27th International Symposium on Theoretical Aspects of Computer Science (STACS) held at Nancy, France during March 6-8, 2010.

†Department of Computer Science and Engineering, IIT Kanpur, India. Email : sbaswana@cse.iitk.ac.in. This work was supported by Research I Foundation, CSE, IIT Kanpur and by Indo-German Max Planck Center for Computer Science (IMPECS).

‡Oracle India Pvt. Ltd., Bangalore - 560029, India. Email : neelesh.khanna@gmail.com.

1 Introduction

The shortest paths problem is a classical and well studied algorithmic problem of computer science. Let $G = (V, E)$ be a given directed weighted graph on $n = |V|$ vertices and $m = |E|$ edges. This problem requires processing of G to compute a data structure which can report shortest path or distance between any two vertices. Two well known and thoroughly studied versions of this problem are the single source shortest paths (SSSP) problem and the all-pairs shortest paths (APSP) problem. For any set $S \subset V$, let $G \backslash S$ denote the graph G after removing all vertices of set S from it. Consider the following extension of the shortest paths problem.

Given a graph $G = (V, E)$ and a small integer $\ell > 1$, construct a compact data structure which, for any set S of at most ℓ vertices, and any $u, v \in V$, can efficiently report the shortest path (or distance) from u to v in $G \backslash S$.

We may denote this path by $\mathcal{P}(u, v, S)$ and the corresponding distance by $\delta(u, v, S)$. The set S may represent the set of *failed* vertices in the graph at any moment, and the path $\mathcal{P}(u, v, S)$ is the shortest path from u to v *avoiding* these failed vertices at that moment. It is required that each such query gets answered in optimal time: $\delta(u, v, S)$ should be reported in $O(1)$ time and the path $\mathcal{P}(u, v, S)$ should be reported in time which is of the order of the number of edges lying on the path.

An ideal goal would be to understand the complexity of the above problem for any arbitrary value of ℓ. However, the first natural step in this direction would be to thoroughly understand the complexity of the case $\ell = 1$, that is, the shortest paths avoiding any single failed vertex. We focus on the single source and all-pairs versions of this problem for undirected unweighted graphs. We show that we can design extremely compact data structures for these versions at the expense of approximation, that is, reporting approximate shortest paths avoiding any failed vertex.

Motivation

The problem of shortest paths avoiding a failed vertex is a very natural extension of the classical shortest paths problem. This fact provides a justification for a thorough study of this problem from theoretical perspective. Moreover, this problem is also motivated by certain applications as follows.

Almost all real life networks which require efficient solution of shortest paths are prone to failure of nodes (vertices) and/or links (edges). So these networks have to have some efficient way of reporting shortest paths avoiding the set of failed nodes (or links) at any given moment. Though these networks are never immune to failures, it is also a fact that the failures are quite infrequent in normal circumstances. Moreover, a failed node (or link) does not remain failed indefinitely; instead it revives and become active after some time due to some repair mechanism which is usually present in such networks. These features can be modeled as follows. There will be at most ℓ failed nodes at any moment of time for some $\ell \ll n$. However, the set of failed nodes may keep changing as the time progresses: The old failed nodes may become active while some active nodes may fail such that the number of failed nodes at any moment is at most ℓ. A typical shortest path query in such networks would be the following. Given any subset S of at most ℓ failed nodes, and any $u, v \in V$, report the shortest path (or distance) from u to v in $G \backslash S$. Moreover, each such query has to be answered as quickly as possible. In particular, once a node fails, we should be able to quickly report the *new* shortest paths to any affected destination. This requirement seems quite natural, especially in the communication networks, where any delay in reporting the new shortest path may lead to congestion due to the queues built up by the packets whose shortest path has changed.

It follows from the discussion above that in addition to being a problem of independent interest from theoretical perspective, the problem of shortest paths avoiding a failed vertex is motivated by certain practical applications as well. This problem is also closely related to the replacement paths problem [4, 16, 19, 21], the most vital node of a shortest path [20], and the k shortest simple paths problem [18, 25] which have been studied quite extensively.

Past work and the need of approximation

Consider the problem of single source shortest paths avoiding any single failed vertex. Let r be a designated source vertex. A trivial data structure for this problem is the following. For each vertex

$x \in V$, compute and store the shortest paths tree at r in the graph $G\backslash\{x\}$. The size of this data structure will be $\Theta(n^2)$. Demetrescu et al. [9] proved a worst case lower bound of $\Omega(m)$ on the size of any data structure for the problem of single source shortest paths avoiding any failed vertex. m can be as large as $\Omega(n^2)$. Hence, for the single source version, in the worst case, the trivial $O(n^2)$ upper bound on the size of any data structure is also asymptotically the best one can hope for.

Interestingly, there are better results known for the all-pairs version of the problem of shortest paths avoiding a failed vertex. The first significant breakthrough on this problem was made by Demetrescu et al. [9]. They designed an $O(n^2 \log n)$ space data structure for a given directed weighted graph, namely *distance sensitivity oracle*. This data structure is capable of reporting the distance (as well as the shortest path) between any two vertices avoiding any single failed vertex in $O(1)$ time. The preprocessing time of this data structure is $O(mn^2)$. Recently, Bernstein and Karger [5] improved the preprocessing time to $O(mn \log n)$.

The quadratic lower bound on the space complexity of the single source version of the problem imposes severe limitations on its solution in practice. This inspires us to explore whether we can achieve subquadratic or near linear size data structure at the expense of reporting approximate shortest path from the source avoiding any failed vertex. Even for the all-pairs version of the problem, the $O(n^2 \log n)$ bound on the size of the data structure, though nearly optimal, is too large for many graphs which appear in various large scale applications [24]. In most of these graphs it is usual to have $m \ll n^2$, hence a table of $\Theta(n^2)$ size may be too large. In fact, due to the same reasons, many algorithms and data structures have been designed for the all-pairs approximate shortest paths problem (without failures) in undirected graphs (see [1, 2, 3, 7, 10, 24]). The prime motivation underlying the design of these algorithms has been to achieve subquadratic space and/or subcubic preprocessing time. However, no data structure has yet been designed for efficiently reporting approximate shortest paths avoiding any failed vertex.

1.1 New results and overview of techniques

We present compact data structures for undirected unweighted graphs which are capable of efficiently reporting approximate shortest path (or distance) between any two vertices avoiding any failed vertex. A path between $u, v \in V$ is said to be t-approximate shortest path if its length is at most t times that of the shortest path between the two. The factor t is usually called the stretch factor. Using new ideas combined with the existing results and techniques, we provide efficient construction of such data structures for both the single source as well as all-pairs versions of this problem. The time taken by our data structures to answer any approximate shortest path or distance query is optimal up to a constant factor. The most impressive feature of our data structures is their *nearly* optimal size. In fact, the size of each of them almost matches the size of the corresponding data structure with no failures.

Single source approximate shortest paths avoiding any failed vertex

First we consider any undirected graph with nonnegative edge weights. For such a graph and a source vertex r, we present an $O(m \log n + n \log^2 n)$ time computable data structure of size $O(n \log n)$. This data structure can report a 3-approximate shortest path from r to any vertex $v \in V$ avoiding any $x \in V$. We then consider undirected unweighted graphs and augment this data structure with some extra information. As a result, this new data structure can report a $(1+\epsilon)$-approximate shortest path from the source to any vertex avoiding any failed vertex for any given $\epsilon > 0$. The space occupied by the data structure is $O(n/\epsilon^3 + n \log n)$.

In order to achieve compact space of these data structures, we proceed as follows. Let P be any path in the shortest paths tree rooted at the source vertex in G. First we present a compact data structure for reporting approximate shortest paths from the source when the failed vertex belongs to path P. A key feature of this data structure is that it has only a small number of *special* vertices for which the shortest paths from the source avoiding any failed vertex will actually be stored. For reporting approximate shortest path from the source to any vertex v when any vertex fails, v will employ the

data structure associated with one of these *special* vertices lying in its *vicinity*. To facilitate it, the undirectedness of the graph is used very crucially.

Once we have an efficient data structure for handling failure of any vertex on a given path P, we extend it to handle failure of any arbitrary vertex using the *heavy path decomposition* technique given Sleator and Tarjan [22]. Using this technique, we break the shortest path tree rooted at the source into vertex disjoint paths and build the data structure mentioned above for each such path. This extension is quite similar in spirit to any divide and conquer approach. In this manner we get our data structure for the single source approximate shortest paths avoiding any failed vertex.

An outcome of independent interest from our data structure is the computation of a subgraph of $O(n/\epsilon^3 + n \log n)$ edges such that each shortest path from the source to any vertex avoiding any single failed vertex is $(1+\epsilon)$-approximated in this subgraph.

All-pairs approximate shortest paths oracle avoiding any single failed vertex

Among the existing data structures for all-pairs approximate shortest paths in undirected graphs [1, 2, 3, 7, 10, 24], the *approximate distance oracle* of Thorup and Zwick [24] stands out due to its amazing features. This data structure, in true sense, is a milestone in the area of all-pairs approximate shortest paths. Thorup and Zwick [24] showed that any undirected graph with nonnegative edge weights can be preprocessed in subcubic time to build a data structure of size $O(kn^{1+1/k})$ for any $k > 1$. This data structure, despite of its subquadratic size, is capable of reporting $(2k-1)$-approximate distance between any two vertices in $O(k)$ time, and hence the name *oracle*. The corresponding approximate shortest path can also be reported in time which is of the order of the number of edges on the path. The size-stretch trade off achieved by the oracle is essentially optimal assuming the 48 year old girth conjecture of Erdős [12].

It is a very natural question to explore if we can design all-pairs approximate distance oracles which may handle any single vertex failure. We show that it is indeed possible for undirected unweighted graphs. For this purpose, we use a couple of new ideas on top of the existing approximate distance oracle of Thorup and Zwick [24]. There are two basic structures, namely *ball* and *cluster*, which form the building blocks of the approximate distance oracle of Thorup and Zwick [24]. We introduce an ϵ-trimming of these structures. Using the ϵ-trimming and a simple inverse relationship between balls and clusters, we succeed in making the approximate distance oracle of Thorup and Zwick [24] robust enough to handle any single vertex failure. Interestingly, we are able to preserve the original trade-off between the space and the stretch as well. In precise words, we achieve the following result here. For any $\epsilon > 0$ and integer $k > 1$, we can preprocess a given undirected unweighted graph to build a data structure of size $O(\frac{k^5}{\epsilon^4} n^{1+\frac{1}{k}} \log^3 n)$. This data structure is capable of answering a query about the shortest path between any two vertices avoiding any single failed vertex with a guarantee of $(2k-1)(1+\epsilon)$ on the stretch. The query time is optimal up to a constant factor.

Recently and independently, Chechik et al. [6] solved quite a similar problem for edge failures. They showed that for any undirected graph with nonnegative edge weights and any integer $f > 0$, a data structure can be built which can report an approximate shortest path between any two vertices given any set of at most f failed edges. The stretch of the reported path is at most $(8k-2)(f+1)$, and the size of the data structure is $O(fkn^{1+1/k} \log nW)$ where W is the ratio of the maximum to the minimum edge weight in the given graph. Though the starting point of their data structure is the approximate distance oracle of Thorup and Zwick [24], they use many new ideas which are quite different from ours.

1.2 Related work

Two well researched problems related to the results of this paper are the replacement paths problem and the k shortest paths problem. Both these problems have primarily been studied for a single source destination pair (s,t). Let P be the shortest path from s to t. The replacement paths problem aims to compute the shortest s-t path avoiding any edge e lying on the path P. The k shortest paths problem aims to compute the k shortest simple paths from s to t. These two problems have a lot of similarity. For example, the second shortest path from s to t will be one of the replacement paths. In fact, as

shown by Yen [25] and Lawler [18], an $O(T(n))$ time algorithm for the replacement paths problem implies an $O(kT(n))$ time algorithm for the k shortest paths problem.

Many efficient algorithms have been designed for replacement paths problem in certain classes of graph [15, 19, 21]. However, there has not been any $o(mn)$ time algorithm till date for this problem in general graphs. Hershberger et al. [16] proved a lower bound of $\Omega(m\sqrt{n})$ for the replacement paths problem and the k shortest paths problem in the path comparison model given by Karger et al. [17]. So here again, approximation seems to be a natural direction of research. Recently, Bernstein [4] designed an algorithm which can report $(1+\epsilon)$-approximation of every replacement path for a given source destination pair. The running time of the algorithm is $O(m\log(nC/c)/\epsilon)$ where C and c are respectively the maximum and minimum edge weights in the graph.

As described earlier, a practical motivation for the shortest paths problem avoiding vertex failure is to study the dynamic shortest paths problem in real life graphs and networks. There has been extensive research on the dynamic shortest path problem in the following model which is quite different from the one we described. There is an initial graph followed by an on-line sequence of insertion and deletion of edges interspersed with shortest path (or distance) queries. Each query has to be answered with respect to the graph which exists at that moment (incorporating all the updates preceding the query). The algorithmic objective here is to maintain a data structure which can answer any distance query efficiently, and can be updated after any edge insertion or deletion in an efficient manner. In particular, the complexity of the algorithm that updates this data structure should be significantly smaller than that of the best static algorithm. Many novel algorithms have been designed in the last ten years for the dynamic shortest paths problem in this model (see [8] and the references therein). On one hand, this model is important since it captures the worst possible hardness of any dynamic graph problem. However, on the other hand, it can also be considered as a very pessimistic model for the dynamic shortest paths problem in real life networks.

1.3 Organization

Section 2 provides various notations and lemmas to be used throughout this paper. Section 3 and 4 are devoted to the data structures for single source approximate shortest paths avoiding any failed vertex. Section 5 is devoted to all-pairs approximate shortest paths oracle avoiding any failed vertex. Like all the previous algorithms for single vertex failure, our algorithms can be easily adapted for handling single edge failure as well without any asymptotic increase in the space and time complexity.

2 Preliminaries

We use the following notations in the context of a given undirected graph $G=(V,E)$ with $n=|V|$, $m=|E|$ and a weight function $\omega : E \rightarrow \mathbb{R}^+$. We shall use r to denote the source vertex.

- T_r : the shortest paths tree rooted at r. Though the underlying graph is undirected, yet it helps conceptually to view each edge of T_r directed away from r.
- $T_r(x)$: the subtree of T_r rooted at x.
- $\mathcal{P}(x,y)$: the shortest path between x and y.
- $\textsc{succ}(v,Q)$: successor of v on path Q. The path Q under consideration will usually be a subpath in T_r, and hence $\textsc{succ}(v,Q)$ will be uniquely defined. [We shall omit the symbol Q from $\textsc{succ}(v,Q)$ when the path Q under consideration is known from the context.]
- $\delta(x,y)$: the length of the shortest path between x and y. For any subset B of vertices, we define $\delta(x,B)$ as $\min_{y\in B}\delta(x,y)$.
- $\mathcal{P}(x,y,z)$: the shortest path between x and y avoiding vertex z.
- $\delta(x,y,z)$: the length of the shortest path between x and y avoiding vertex z. For any subset B of vertices, we define $\delta(x,B,z)$ as $\min_{y\in B}\delta(x,y,z)$.

- $G_r(x)$: the subgraph induced by the vertices of $T_r(x)\backslash\{x\}$ and augmented by vertex r and edges from r as follows. For each $v \in T_r(x), v \neq x$ with neighbors outside $T_r(x)$, keep an edge (r, v) of weight $= \min_{(u,v)\in E, u\notin T_r(x)}(\delta(r, u) + \omega(u, v))$.
- $P \cdot Q$: a path formed by concatenating path Q at the end of path P with an edge $(u, v) \in E$, where u is the last vertex of P and v is the first vertex of Q.
- $E(X)$: the set of edges from E with at least one endpoint in X.

We now state a couple of properties and terminologies related to $\mathcal{P}(r, v, x)$. These will be used in a crucial manner in this paper.

1. **optimal subpath property**
   Each subpath of $\mathcal{P}(r, v, x)$ is also the shortest path between its endpoints upon failure of x.
2. **triangle inequality**
   For each $x, v, z \in V$, $\quad \delta(r, v, x) \leq \delta(r, z, x) + \delta(z, v, x)$
3. **detour**
   Path $\mathcal{P}(r, v, x)$ must leave the path $\mathcal{P}(r, v)$ at some vertex before x, say a, and join it back at some vertex after x, say b. The subpath of $\mathcal{P}(r, v, x)$ between vertex a and b, say $p_{a,b}$, will intersect $\mathcal{P}(r, v)$ at exactly two vertices, namely a and b. This path $p_{a,b}$ is called the detour associated with $\mathcal{P}(r, v, x)$.

The following important observation follows immediately from the optimal subpath property.

Observation 2.1 *[9] Let x be any vertex in T_r, and let v be any vertex belonging to the subtree $T_r(x)$. The path $\mathcal{P}(r, v, x)$ must be of the form $A \cdot B$, where A is a path present in $T_r\backslash T_r(x)$ and B is a path present in the subgraph of G induced by $T_r(x)\backslash\{x\}$.*

Observation 2.1 and the definition of $G_r(x)$ given above leads to the following lemma immediately.

Lemma 2.1 *In order to compute shortest paths from r avoiding x, it suffices to perform Dijkstra's algorithm from vertex r in the graph $G_r(x)$.*

Lemma 2.1 leads to the following result for unweighted graph when the failed vertex lies within *small* distance from the source. A generalized version of this result was used by Demetrescu et al. [9] in efficient construction of all-pairs distance sensitivity oracle.

Lemma 2.2 *Consider an unweighted graph $G = (V, E)$ and a vertex $r \in V$. For any integer t, we can preprocess the graph in $O(mt + nt \log n)$ time to build a data-structure of size $O(nt)$ which can answer shortest-path/distance queries from r to any vertex upon failure of any single vertex within distance t from r.*

Proof: Observe that for unweighted graph, the shortest path tree T_r is the same as the breadth first search (BFS) tree. Let $x_1, \ldots, x_j$ be the vertices lying at any level $\ell \leq t$ in T_r. To prove the lemma, it will suffice if we can construct an $O(n)$ space data structure which can report $\mathcal{P}(r, v, x_i)$ for any $v \in V, 1 \leq i \leq j$. We employ Lemma 2.1 to design such a data structure. First note that each of the subtrees $\{T_r(x_i) \mid 1 \leq i \leq j\}$ are vertex disjoint. For each $1 \leq i \leq j$, we build and store a shortest path tree rooted at r in the graph $G_r(x_i)$. These shortest path trees together with T_r can report $\mathcal{P}(r, v, x_i)$ for any $v \in V, 1 \leq i \leq j$ in optimal time. It can be observed that the total space occupied by these trees will be $O(n)$ and their total computation time will be $O(m + n \log n)$. •

Our data structures will also use an efficient data structure for answering lowest common ancestor (LCA) queries on T_r.

Lemma 2.3 *[14] A rooted tree on n vertices can be preprocessed in $O(n)$ time to build a data structure of size $O(n)$ which can report, for any two vertices u, v, their lowest common ancestor in $O(1)$ time.*

For the sake of simplicity, we shall assume that $\mathcal{P}(u,v,x)$ exists for every $u,v,x \in V$. In other words, we assume that the given graph is biconnected. However, we can handle graphs which are not necessarily biconnected, essentially by breaking the graph into maximal biconnected components and then solving the problem of approximate shortest paths avoiding vertex failure for each biconnected component. We also assume that $\mathcal{P}(u,v,x)$ is unique. If there are multiple shortest paths between u and v in $G\backslash\{x\}$, we may declare any one of these shortest paths as $\mathcal{P}(u,v,x)$.

3 Single source 3-approximate shortest paths avoiding any failed vertex

In this section we design an $O(n \log n)$ space data structure for any undirected graph with nonnegative edge weights. This data structure is capable of reporting 3-approximate shortest path from a designated source r to any vertex $v \in V$ whenever there is any single vertex failure in the graph.

First, as a warm up, we describe a simple idea for achieving 3-approximation of distance from source r to every vertex avoiding any single failed vertex. Let $x \in V$ be the failed vertex at a moment and $v_1, ..., v_j$ be its children in T_r as shown in Figure 1. It is easy to observe that the failure of vertex

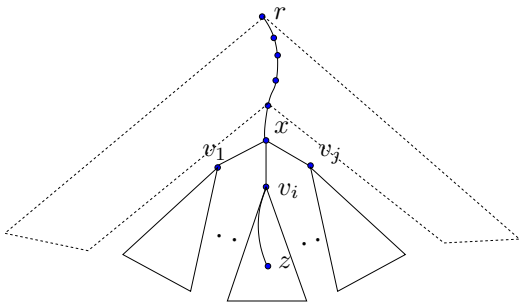

Figure 1: Storing distances $\delta(r, v_i, x)$ suffices to retrieve 3-approximation of $\delta(r,z,x)$ for any $z \in T_r(v_i)$

x may alter the distance from r to vertices belonging to $T_r(v_i), 1 \leq i \leq j$ only. Consider any vertex v_i and $z \in T_r(v_i)$. Note that the shortest path $\mathcal{P}(v_i,z)$ remains intact even after removal of x, and its length is certainly less than $\delta(r,z)$. So, in order to travel from r to z when x fails, we may first travel along shortest route to v_i (that is $\mathcal{P}(r,v_i,x)$) and then along $\mathcal{P}(v_i,z)$. The distance traveled in this manner won't be *too large* compared to the distance associated with $\mathcal{P}(r,z,x)$. In fact, exploiting undirectedness of the graph and the triangle inequality property, we can show that $\delta(r,v_i,x)+\delta(z,v_i)$ is 3-approximation of $\delta(r,z,x)$ as follows.

$$\begin{aligned}\delta(r,v_i,x)+\delta(z,v_i) &\leq \delta(r,z,x)+\delta(z,v_i,x)+\delta(z,v_i)\\ &\leq \delta(r,z,x)+2\delta(z,v_i)\\ &\leq \delta(r,z,x)+2\delta(r,z) \leq 3\delta(r,z,x)\end{aligned}$$

Remark 3.1: The approximation factor may be much smaller than 3 in case $\delta(v_i,z) \ll \delta(r,z)$. We shall employ this observation carefully in the next section to design a data structure for unweighted graphs which can report $(1+\epsilon)$-approximate shortest paths from source r avoiding any failed vertex.

Therefore, based on the above discussion, storing $\delta(r,v_i,x)$ for all $i \leq j$ suffices to retrieve 3-approximate distance from r to any vertex in the graph whenever vertex x fails. Processing each $x \in T_r$ in this manner leads to a data structure of $O(n)$ space which can report 3-approximate distance from r avoiding any single failed vertex. However, to extract the corresponding approximate shortest path efficiently would require storing the paths $\mathcal{P}(r,v_i,x)$ for each v_i. Since each of these paths might be quite long and different from all other paths, this approach may lead to $\Theta(n^2)$ space in the worst case. So the challenging task is to design a data structure which occupies *nearly* $O(n)$ space, and yet allows efficient retrieval of 3-approximate shortest paths from r whenever any single vertex fails. To achieve this objective we first solve a simpler subproblem where the failing vertex belongs to a given path $Q \in T_r$. Later we use divide and conquer strategy to solve our main problem.

3.1 Subproblem: Handling failure of a vertex lying on a given path in T_r

Given the shortest path tree T_r, let Q be any path in T_r from some vertex q to some vertex t. Without loss of generality assume that t is a leaf node in T_r. Otherwise, we can always extend Q to some leaf node. We shall design a data structure which can efficiently report a 3-approximate shortest path from r to any $v \in V$ when some vertex from Q fails. This data structure is inspired by the algorithm of Nardelli et al. [20] for computing the most vital vertex on a shortest path. Consider any $x \in Q, x \neq t$. We may partition the tree $T_r \backslash \{x\}$ into the following 3 parts as shown in Figure 2.

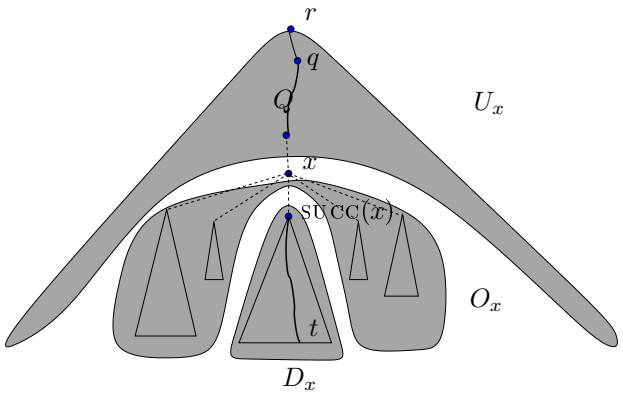

Figure 2: Partitioning of the shortest path tree T_r at $x \in Q$

1. U_x : the tree T_r after removing the subtree $T_r(x)$
2. D_x : the subtree of T_r rooted at $succ(x)$
3. O_x : the portion of T_r left after removing U_x, x, and D_x.

It can be observed that whenever a vertex $x \in Q$ fails, the shortest path and distance from source may change only for the vertices of set D_x and O_x. Based on this observation, the data structure will actually consist of the following two data structures.

1. $H_d(Q)$: the data structure to report 3-approximate shortest path from r to any $v \in D_x$ when any vertex $x \in Q$ fails.
2. $H_o(Q)$: the data structure to report 3-approximate shortest path from r to any $v \in O_x$ when any vertex $x \in Q$ fails.

Now we describe the above two data structures and their preprocessing algorithms. Let $Q = \langle q(= x_0), x_1, ..., x_k(= t)\rangle$ be the given path. In the following discussion, we shall use U_i, D_i, O_i as succinct notations for $U_{x_i}, D_{x_i}, O_{x_i}$ respectively.

3.1.1 Description and preprocessing of the data structure $H_d(Q)$

The data structure H_d will report 3-approximate shortest path to any $v \in D_i$ when x_i fails for any $i < k$. To achieve this objective, it will store distance $\delta(r, x_{i+1}, x_i)$ and the corresponding path $\mathcal{P}(r, x_{i+1}, x_i)$ for each $i < k$. However, it will store these paths implicitly so that the overall space occupied by H_d will be $O(|T_r(x_0)|)$. Replying to a query for $\mathcal{P}(r, v, x_i)$, for any $v \in D_i$, it will report $\mathcal{P}(r, x_{i+1}, x_i) \cdot \mathcal{P}(x_{i+1}, v)$. It follows from the discussion in the beginning of this section that this path will be a stretch-3 approximation of $\mathcal{P}(r, v, x_i)$. To achieve efficient computation and compact storage of $\mathcal{P}(r, x_{i+1}, x_i)$ for all $0 \leq i < k$, we exploit the following lemma.

Lemma 3.2 *The shortest path $\mathcal{P}(r, x_{i+1}, x_i)$ is of the form $P_1 \cdot P_2$ where P_1 is a shortest path from r in the subgraph induced by $U_i \cup O_i$, and P_2 is a path present in $T_r(x_{i+1})$.*

Proof: Let z be the first vertex of the path $\mathcal{P}(r, x_{i+1}, x_i)$ which belongs to D_i. Define P_1 as the portion of $\mathcal{P}(r, x_{i+1}, x_i)$ preceding z, and P_2 as the portion starting from z. All the vertices of P_1 belong to $U_i \cup O_i$. So P_1 is certainly a shortest path from r in the subgraph induced by $U_i \cup O_i$. It

follows from the optimal subpath property and undirectedness of the graph that $P_2 = \mathcal{P}(x_{i+1}, z, x_i)$. However, $\mathcal{P}(x_{i+1}, z, x_i)$ is the same as $\mathcal{P}(x_{i+1}, z)$, and the latter is already present in $T_r(x_{i+1})$. •

It follows from Lemma 3.2 that in order to compute $\mathcal{P}(r, x_{i+1}, x_i)$, first we need to compute shortest paths from r in the subgraph induced by $U_i \cup O_i$. Note that the shortest paths from r to all vertices of U_i in this subgraph are the same as in the original graph, and are already present in T_r. So we just need to compute shortest paths from r to vertices of O_i in this subgraph. We do it by executing Dijkstra's algorithm from r in the subgraph induced by vertices $O_i \cup \{r\}$ and the following additional edges. For each $o \in O_i$ with at least one neighbor in U_i, we add an edge (r, o) with weight $= \min_{(u,o)\in E, u\in U_i}(\delta(r, u) + \omega(u, o))$. Let τ_r^i denote the shortest path tree computed in this manner, and let $\delta_i(r, v)$ denote the distance from r to any $v \in U_i \cup O_i$ in the subgraph induced by $U_i \cup O_i$. It follows from Lemma 3.2 that the first vertex on $\mathcal{P}(r, x_{i+1}, x_i)$ which belongs to D_i is the vertex z which minimizes $\delta_i(r, y) + \omega(y, z) + \delta(x_{i+1}, z)$ over all $(y, z) \in E$ with $y \in U_i \cup O_i, z \in D_i$. Let us denote this vertex by z_i, and let $y_i \in U_i \cup O_i$ be the vertex which precedes z_i on the path $\mathcal{P}(r, x_{i+1}, x_i)$. It can be observed that the entire shortest path $\mathcal{P}(r, x_{i+1}, x_i)$ can be reported using T_r, τ_r^i, and the edge (y_i, z_i) in time of the order of number of edges on $\mathcal{P}(r, x_{i+1}, x_i)$.

Data structure $H_d(Q)$. Based on the above discussion, the data structure H_d will store only the edge (y_i, z_i) and the tree τ_r^i for each $i < k$. Due to mutual disjointness of O_i's, it follows that the space required by H_d will be of the order of $\sum_{i<k} |\tau_r^i| = O(|T_r(q)|)$.

Efficient computation of $H_d(Q)$. The time required to build τ_r^i's for all $i < k$ will be of the order of $\sum_{v\in T_r(q)}(\deg(v) + \log n)$ due to mutual disjointness of O_i's. The only extra computational task is the computation of edges (y_i, z_i) for all $i < k$ which we can perform efficiently as follows.

The key idea used is that (U_i, D_i, O_i) has a lot of overlap with $(U_{i+1}, O_{i+1}, D_{i+1})$. This overlap can be exploited to compute all the edges $\{(y_i, z_i)|0 < i < k\}$ efficiently in an incremental fashion as follows. We keep a heap data structure storing vertices. At the time of computation of (y_i, z_i), the heap consists of vertices of set D_i and the key of each vertex $z \in D_i$ is defined as

$$key(z) = \min_{(y,z)\in E, y\in U_i\cup O_i} (\delta_i(r, y) + \omega(y, z) + \delta(r, z))$$

With each $key(z)$, we also store the corresponding edge (y, z) which minimizes the value of $key(z)$ as defined above. It can be seen that z_i corresponds to the vertex in the heap with the smallest key. So the computation of z_i (and hence the edge (y_i, z_i) as well) just requires a FIND_MIN operation on the heap. Now observe that $D_{i+1} = D_i \backslash (O_{i+1} \cup \{x_{i+1}\})$. Therefore, for computing z_{i+1} we just need to update the heap as follows.

- DELETE each vertex of set $O_{i+1} \cup \{x_{i+1}\}$ from the HEAP.
- For each edge $(y, z) \in E$ with $y \in O_{i+1}, z \in D_{i+1}$, perform DECREASE_KEY on z as follows.

$$key(z) \leftarrow \min(key(z),\ \delta_{i+1}(r, y) + \omega(y, z) + \delta(r, z))$$

- For each edge $(y, z) \in E$ with $y \in O_i \cup \{x_i\}, z \in D_{i+1}$, perform DECREASE_KEY on z as follows.

$$key(z) \leftarrow \min(key(z),\ \delta(r, y) + \omega(y, z) + \delta(r, z))$$

Performing FIND_MIN operation on the heap will now report (y_{i+1}, z_{i+1}). In this manner, we compute the entire set $\{(y_i, z_i)|0 \le i < k\}$ by performing certain HEAP operations. In particular, there will be $O(k)$ FIND_MIN operations and $O(|T_r(q)|)$ DELETE_KEY operations. To bound the number of DECREASE_KEY operations, note that each of these operations is associated with an edge whose at least one endpoint is in $T_r(q)$. Moreover, it follows from the description given above that there will be at most two DECREASE_KEY operations associated with each such edge. Hence, the total number of DECREASE_KEY operations will be at most $2\sum_{v\in T_r(q)} \deg(v)$. Using Fibonacci heap [13], all these HEAP operations can be performed in $\sum_{v\in T_r(q)}(\deg(v) + \log n)$ time. We can thus conclude the following lemma.

Lemma 3.3 *A shortest path $Q = \mathcal{P}(q,t)$ present in T_r can be preprocessed to build a data structure $H_d(Q)$ of $O(|T_r(q)|)$ size. In case of failure of any $x \in Q$, this data structure can report 3-approximation of $\delta(r,v,x)$ as well as $\mathcal{P}(r,v,x)$ for any $v \in D_x$. The query time is optimal up to a constant factor and the preprocessing time of $H_d(Q)$ is of the order of $\sum_{v \in T_r(q)}(\deg(v) + \log n)$.*

3.1.2 Description and preprocessing of the data structure $H_o(Q)$

The data structure $H_o(Q)$ will report 3-approximate shortest paths to vertices of O_i upon failure of x_i for any $i < k$. The preprocessing of H_o will employ the data structure H_d described above. Recall that H_d can report 3-approximate shortest paths to vertices of D_i upon failure of x_i for any i. Here we shall prove an interesting generic result which states that if we have a data structure to retrieve α-approximate shortest paths from r to vertices of D_i upon failure of x_i, then we can use it to have a data structure to retrieve α-approximate shortest paths to vertices of O_i as well. To prove this result, consider the subgraph induced by $O_i \cup \{r\}$ and some extra edges which are defined as follows.

- For each $o \in O_i$ having neighbors from U_i, add an edge (r,o) and assign it weight equal to $\min_{(u,o)\in E, u \in U_i}(\delta(r,u) + \omega(u,o))$.
- For each $o \in O_i$ having neighbors from D_i, add an edge (r,o) and assign it weight equal to $\min_{(u,o)\in E, u \in D_i}(\hat{\delta}(r,u,x_i) + \omega(u,o))$, where $\hat{\delta}(r,u,x_i)$ is the α-approximate distance to u upon failure of x_i. (In the present situation we have $\alpha = 3$.)

In case of multiple edges introduced from r to o as a result of the above steps, keep the edge with the least weight only. Let us denote this graph by $G(r,O_i)$.

Lemma 3.4 *The shortest paths tree from r in the graph $G(r,O_i)$ will store α-approximate shortest paths from r to all $v \in O_i$ avoiding x_i.*

Proof: Consider the shortest path $\mathcal{P}(r,o,x_i)$ for any $o \in O_i$. If this path does not pass through any vertex of D_i, then it follows from the construction of $G(r,O_i)$ that a path of length exactly equal to $\delta(r,o,x_i)$ is present in the subgraph $G(r,O_i)$ also. Now consider the case when the path $\mathcal{P}(r,o,x_i)$ passes through one or more vertices of D_i. Let (u,v) be the last edge on the path $\mathcal{P}(r,o,x_i)$ such that $u \in D_i$ and $v \in O_i$. Consider the prefix of the shortest path $\mathcal{P}(r,o,x_i)$ ending at v. It follows from optimal subpath property that this prefix is also a shortest path from r to v avoiding x_i, and its length is $\delta(r,u,x_i) + \omega(u,v)$. Now note that the edge (r,v) in the graph $G(r,O_i)$ has weight $\hat{\delta}(r,u,x_i) + \omega(u,v)$ which is bounded by $\alpha\delta(r,u,x_i) + \omega(u,v)$. Hence the prefix of the path $\mathcal{P}(r,o,x_i)$ up to v is stretched by at most α in $G(r,O_i)$. Now the suffix of the path $\mathcal{P}(r,o,x_i)$ following v consists of vertices of set O_i only, and so it is present entirely in the graph $G(r,O_i)$. Hence, the shortest path from r to o in $G(r,O_i)$ is an α-approximation of $\mathcal{P}(r,o,x_i)$. •

The data structure $H_o(Q)$. Based on the above discussion, this data structure stores the shortest path tree built for the graph $G(r,O_i)$ for each $i < k$. It can be seen that this data structure in conjunction with tree T_r and $H_d(Q)$ can report a 3-approximate shortest path from r to any $o \in O_i$ upon failure of x_i for any $i < k$.

Preprocessing of $H_o(Q)$. It will take $O(|E(O_i)| + |O_i|\log|O_i|)$ time to build the shortest path tree on $G(r,O_i)$ using Dijkstra's algorithm. Once again, note that the sets O_i's are mutually disjoint. Therefore, the total space required by $H_o(Q)$ is $O(|T_r(q)|)$. Furthermore, the total time spent in building these shortest path trees for each $i < k$ will be $O(\sum_{v \in T_r(q)}(\deg(v) + \log n))$.

Lemma 3.4 and the above discussion imply the following observation which we shall use later for improving the stretch factor when the graph is unweighted.

Observation 3.1 *Given tree T_r and a path $Q = \mathcal{P}(q,t)$ present in T_r, if there is a data structure which can report $(1+\epsilon)$-approximate shortest paths from r to vertices of D_x upon failure of any $x \in Q$, then we can build a data structure $H_o(Q)$ which can report $(1+\epsilon)$-approximate shortest paths to all vertices of O_x upon failure of any $x \in Q$.*

Query answering: We now show that the data structures $H_d(Q)$ and $H_o(Q)$ together can be used for reporting 3-approximate shortest path from source r to any vertex v whenever any vertex $x_i \in Q$ fails. If $LCA(v, x_i) \neq x_i$, the shortest path from r to v is unaffected by the failure of x_i, so we just report $\mathcal{P}(r, v)$. Otherwise, we determine if $v \in D_i$ or $v \in O_i$. It can be seen that $v \in D_i$ if $LCA(v, x_{i+1}) = x_{i+1}$, and $v \in O_i$ otherwise. If $v \in D_i$ we use $H_d(Q)$, else we use $H_o(Q)$ to report the approximate shortest path between r and v avoiding x_i.

Theorem 3.1 *An undirected weighted graph $G = (V, E)$, a source $r \in V$, and a shortest path $Q = \mathcal{P}(q, t)$ in T_r can be processed to build a data structure which can report 3-approximate shortest path from r to any $v \in V$ upon failure of any single vertex from Q. The size of this data structure is $O(|T_r(q)|)$ and its preprocessing time is of the order of $\sum_{v \in T_r(q)}(\deg(v) + \log n))$.*

3.2 Data structure for handling failure of any vertex in T_r

Now we shall describe a data structure $\mathcal{H}$ for reporting approximate shortest path from r to any vertex $v \in V$ avoiding any failed vertex $x \in T_r$. We take the following approach. Partition the tree into vertex disjoint paths, and for each of these paths build data structure described in the previous section (see Theorem 3.1). However, any arbitrary partitioning of T_r will not lead to efficient construction and compact size of the final data structure. Therefore, we employ a partitioning scheme devised by Sleator and Tarjan [22]. The following lemma lies at the heart of this scheme.

Lemma 3.5 *[22] There exists an $O(n)$ time algorithm to compute a path Q in T_r whose removal splits T_r into a set of disjoint subtrees $T_r(v_1), ..., T_r(v_j)$ such that for each $i \leq j$:*
- *$|T_r(v_i)| < n/2$ and $Q \cap T_r(v_i) = \emptyset$.*
- *$T_r(v_i)$ is connected to Q through some edge for each $i \leq j$.*

Proof: We provide a simple traversal algorithm which computes the path Q and a set $\mathcal{T}$ of subtrees satisfying all the properties mentioned above. Initially $\mathcal{T} = \emptyset$. Let there be ℓ children $x_1, ..., x_\ell$ of the root r. Let $T_r(x_j)$ be the largest subtree among $T_r(x_1), ..., T_r(x_\ell)$. Add every subtree $T_r(x_i), i \neq j$ to $\mathcal{T}$, and traverse the edge (r, x_j). Now from x_j, we traverse the edge to that descendant through which hangs the largest subtree, and add the remaining subtrees to $\mathcal{T}$. Keep on traversing T_r in this manner and stop when we reach a leaf vertex. This defines the path Q. It can be seen that each subtree in the set $\mathcal{T}$ is connected to Q through some edge and has size $< n/2$. This completes the proof. •

Procedure `Partition`(T) employs Lemma 3.5 to compute a partition of any rooted tree T into a set $\mathbb{P}$ of vertex disjoint paths. It is easy to execute this procedure in $O(n \log n)$ time. See Figure 3

Procedure `Partition`(T)

```
if |T| = 1 then return {T};
else
    compute the path Q originating from root in T as described by Lemma 3.5;
    P ← {Q};
    let v_1, ..., v_j be the roots of the subtrees of T directly connected to the path Q through an
    edge;
    foreach 1 ≤ i ≤ j do  P ← P ∪ Partition(T_r(v_i)) ;
    return P ;
```

for a better illustration of this procedure. Each maximal sequence of solid edges represents a path in $\mathbb{P}$. Each dashed edge represents an edge which joins two different paths in $\mathbb{P}$. Moreover, if (x, y) is a dashed edge then it follows from Lemma 3.5 that $|T_r(y)| < \frac{1}{2}|T_r(x)|$. Thus while traversing from root to any leaf of T_r, we shall encounter at most $\log n$ dashed edges. This leads to the following lemma.

Lemma 3.6 *For any vertex v, the path to the root in T_r intersects at most $\log n$ paths in $\mathbb{P}$.*

Data structure $\mathcal{H}$. The data structure $\mathcal{H}$ will consist of two data structures $\mathcal{H}_d$ and $\mathcal{H}_o$ which are computed as follows.

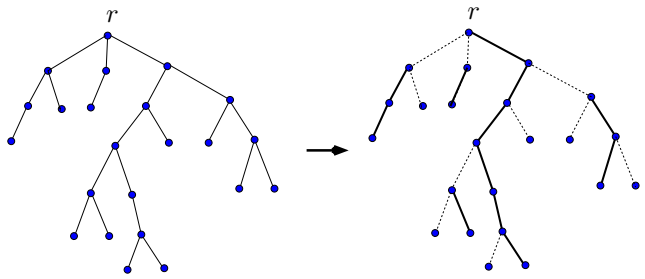

Figure 3: Partitioning of T_r into disjoint paths as computed by $\texttt{Partition}(T_r)$

1. $\mathbb{P} \leftarrow \texttt{Partition}(T_r)$.
2. For each vertex, store pointer to the path in $\mathbb{P}$ to which it belongs.
3. $\mathcal{H}_d \leftarrow \{H_d(Q) \mid Q \in \mathbb{P}\}$; $\mathcal{H}_o \leftarrow \{H_o(Q) \mid Q \in \mathbb{P}\}$.

For reporting 3-approximation of $\mathcal{P}(r, v, x)$ for any $v, x \in V$, first we determine the path $Q \in \mathbb{P}$ to which x belong and then query the data structure $H_d(Q)$ or $H_o(Q)$ accordingly as described earlier.

Analysis of the space and preprocessing time of the data structure.

Consider any path $Q = \mathcal{P}(q, t)$ in the partition $\mathbb{P}$. Theorem 3.1 implies that each vertex $v \in T_r(q)$ contributes $O(1)$ amount to the size and $O(\deg(v)+\log n)$ amount to the preprocessing time of $H(Q)$. Furthermore, it follows from Lemma 3.6 that any vertex v will make this contribution to at most $\log n$ such paths in $\mathbb{P}$. Thus the data structure will have $O(n \log n)$ space and $O(m \log n + n \log^2 n)$ preprocessing time. Hence we can conclude with the following theorem.

Theorem 3.2 *An undirected weighted graph $G = (V, E)$ and a vertex $r \in V$ can be processed in $O(m \log n + n \log^2 n)$ time to build a data structure $\mathcal{H}$ of size $O(n \log n)$. This data structure can report a 3-approximate shortest path from r to any vertex $v \in V$ avoiding any failed vertex $x \in V$ in time which is optimal up to a constant factor.*

4 Single source $(1+\epsilon)$-approximate shortest paths avoiding any failed vertex

The data structure described above can report 3-approximate shortest paths from a given fixed vertex r whenever some vertex in the graph fails. Note that this data structure is actually a collection of basic data structures $H_o(Q)$ and $H_d(Q)$ defined for various paths in the partition $\mathbb{P}$ of T_r. Here, the reader is recommended to recall the dependency of $H_o(Q)$ on $H_d(Q)$ which led to Observation 3.1. Not only the construction of $H_o(Q)$ requires $H_d(Q)$, but the stretch factor associated with $H_o(Q)$ is also defined by the stretch factor associated with $H_d(Q)$. We shall use this fact in a crucial manner. We shall show that for unweighted graphs, it is possible to augment the collection $\mathcal{H}_d = \{H_d(Q) \mid Q \in \mathbb{P}\}$ with supplementary data structures to build a data structure $\mathcal{H}_d^+$ which guarantees a stretch factor of $(1+\epsilon)$ for arbitrarily small ϵ. Now, it is an immediate implication of Observation 3.1 that if we now construct $\mathcal{H}_o = \{H_o(Q) \mid Q \in \mathbb{P}\}$ using $\mathcal{H}_d^+$, the stretch factor associated with $\mathcal{H}_o$ will also be $1+\epsilon$. In this way, $\mathcal{H}_d^+$ and $\mathcal{H}_o$ together will constitute a data structure for reporting $(1+\epsilon)$-approximate shortest paths from r avoiding any failed vertex in the graph. With this overview of our approach, we now provide the key ideas to augment $\mathcal{H}_d$ in order to achieve improved stretch.

Let us first revisit the strategy underlying $\mathcal{H}_d$ which guarantees an approximation factor of 3. Consider failure of any vertex x. Let $Q \in \mathbb{P}$ be the path to which x belongs. For reporting approximate distance between r and $v \in D_x$ when x fails, the data structure $\mathcal{H}_d$ employs $H_d(Q)$. The approximate distance reported is $\delta(r, \textsc{succ}(x), x) + \delta(\textsc{succ}(x), v)$ which is bounded by $\delta(r, v, x) + 2\delta(\textsc{succ}(x), v)$. (Here $\textsc{succ}(x) = \textsc{succ}(x, Q)$ is the successor of x on path Q.) Hence the stretch is

$$\frac{\delta(r, v, x) + 2\delta(\textsc{succ}(x), v)}{\delta(r, v, x)} \tag{1}$$

Though the above stretch is bounded by 3 in the worst case, it is bounded by $(1+\epsilon)$ for any $\epsilon > 0$ if the following condition holds.

$\mathcal{C}$: SUCC(x) is *close* to v, that is, $\delta(\text{SUCC}(x), v) \leq \frac{\epsilon}{2}\delta(r, v)$.

Whenever the condition $\mathcal{C}$ does not hold, we shall ensure that there will be some ancestor w of v lying on $\mathcal{P}(x, v)$, called a *special* vertex, satisfying the following two properties.

1. $\delta(w, v) \ll \delta(r, v)$, that is w is much closer to v than r.
2. Vertex w stores approximate shortest path to r avoiding x (with the approximation factor arbitrarily close to 1).

These two properties will ensure that whenever condition $\mathcal{C}$ does not hold, vertex v may query its special vertex w first to retrieve the approximate shortest path from r to w avoiding x. This path is concatenated with the path $\mathcal{P}(w, v)$ which remains intact when x fails. The resulting path will turn out to be $(1+\epsilon)$-approximation of $\mathcal{P}(r, v, x)$ for any desired $\epsilon > 0$. The data structure $\mathcal{H}_d^+$ will be just the union of $\mathcal{H}_d$ and the supplementary data structures associated with each special vertex.

We shall first describe the construction of the set of special vertices in T_r. Note that T_r is identical to the breadth first search (BFS) tree rooted at r. We shall use LEVEL(v) to denote the level (or distance from r) of vertex v in T_r. After defining the set of special vertices, we shall describe the data structure stored for each special vertex. However, before all this, we would like to address a minor technical point. We shall employ Lemma 2.2 to handle the failure of any vertex which lies up to level $\ell_0 = \Theta(\log n)$ in T_r. This will require $O(m \log n + n \log^2 n)$ time and $O(n \log n)$ space. So, henceforth we shall focus on the failure of only those vertices in T_r which lie at level $> \ell_0$.

4.1 Constructing the set of special vertices

Let h be the height of BFS tree rooted at r. Let us introduce a variable $\epsilon' < 1$ whose value will be defined later in terms of ϵ. Without loss of generality assume that $\ell_0 = \lfloor (1+\epsilon')^{i_0} \rfloor$ for some i_0. We now describe the construction of the set of special vertices.

Let L be the set of positive integers defined as $L = \{i \mid \ell_0 \leq \lfloor (1+\epsilon')^i \rfloor < h\}$. For a given $i \in L$, let us define a subset S_i as

$$S_i = \{u \in V \mid \text{LEVEL}(u) = \lfloor (1+\epsilon')^i \rfloor \ \textbf{and} \ |T_r(u)| \geq \epsilon' \text{LEVEL}(u)\}$$

The set of special vertices is $S = \cup_{i \in L} S_i$. We introduce two terminologies in the context of these special vertices.

- For any vertex $v \in V$, $S(v)$ denotes the nearest ancestor of v which belongs to set S. (In case $v \in S$, then $S(v) = v$.)
- For a vertex $u \in S$, $V(u)$ denotes the set of vertices $v \in V$ with $S(v) = u$. In essence, the vertex u will serve as the special vertex for each vertex from $V(u)$. Upon failure of any vertex $x \in \mathcal{P}(r, u)$, each vertex of set $V(u)$ may access the data structure stored at u for retrieval of approximate shortest path/distance from the source.

Observation 4.1 *If a special vertex u lies at level ℓ, then there are at least $\epsilon'\ell$ vertices in $V(u)$.*

Figure 4 provides a description of the special vertices and the set $V(u)$ in tree T_r. The following lemma states that each vertex v is *much closer* to $S(v)$ than the source vertex.

Lemma 4.1 *Let $v \in V \backslash S$, then $\delta(v, S(v)) \leq \left(\frac{2\epsilon'}{1+\epsilon'}\right)$LEVEL$(v)$.*

Proof: Let ℓ be the level of v in T_r. Then, there must be an $i \in L$ such that $\lfloor (1+\epsilon')^i \rfloor < \ell \leq \lfloor (1+\epsilon')^{i+1} \rfloor$. Let a be the ancestor of v at level $\lfloor (1+\epsilon')^i \rfloor$. If $S(v) = a$, then it can be observed that $\delta(v, S(v)) = \ell - \lfloor (1+\epsilon')^i \rfloor \leq \ell - \frac{\ell}{1+\epsilon'} = \frac{\epsilon'\ell}{1+\epsilon'}$. Else, let b be the ancestor of v at level $\lfloor (1+\epsilon')^{i-1} \rfloor$. Clearly $|T_r(b)| \geq \epsilon' \lfloor (1+\epsilon')^{i-1} \rfloor$. Thus $b \in S$ and $S(v) = b$. Now $\delta(v, S(v)) \leq \ell - \lfloor (1+\epsilon')^{i-1} \rfloor \leq \ell(1 - \frac{1}{(1+\epsilon')^2}) \leq \frac{2\epsilon'\ell}{1+\epsilon'}$. •

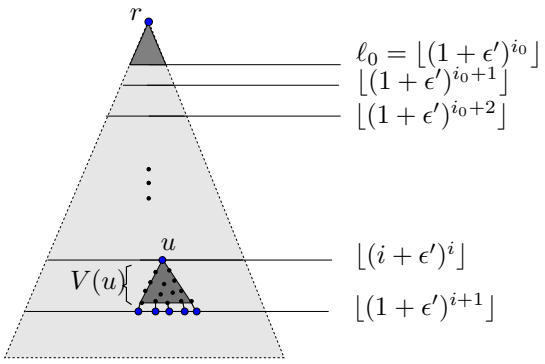

Figure 4: Splitting the tree T_r into geometrically increasing levels to constitute the special vertices.

4.2 The data structure for a special vertex

We shall process the special vertices in a top down fashion in T_r while constructing the data structure associated with them. Consider a special vertex v with $\text{LEVEL}(v) = \lfloor (1+\epsilon')^i \rfloor$ and $i > i_0$. (Recall that for special vertices at level $\ell_0 = \lfloor (1+\epsilon')^{i_0} \rfloor$, we already store exact shortest path to r upon failure of any vertex). We shall now describe a compact data structure to be stored at v which will facilitate retrieval of $(1+2\epsilon')$-approximate shortest path from r to v upon failure of any vertex $x \in \mathcal{P}(r,v)$. The vertex v will store the corresponding path in a field $\text{PATH}(v,x)$.

Let v' be the special vertex which is present at level $\lfloor (1+\epsilon')^{i-1} \rfloor$ and is ancestor of v. The data structure stored at v will be defined in terms of various cases of the failing vertex $x \in \mathcal{P}(r,v)$ as follows.

If $x \in \mathcal{P}(v',v)$, then consider the path $\mathcal{P}(r,\text{SUCC}(x),x) \cdot \mathcal{P}(\text{SUCC}(x),v)$ which is already available in $\mathcal{H}_d$. It follows from Equation 1 that this path is $(1+2\epsilon')$-approximation of $\mathcal{P}(r,v,x)$. So $\text{PATH}(v,x)$ may store this path implicitly by keeping a pointer to $\mathcal{P}(r,\text{SUCC}(x),x)$ stored in $\mathcal{H}_d$. Hence we require only $O(1)$ extra storage in this case.

Let us now consider the nontrivial case when $x \in \mathcal{P}(r,v')$ and $x \neq v'$. Let $p_{a,b}$ be the detour associated with $\mathcal{P}(r,v,x)$. This detour can be of any of the following two types as shown in Figure 5.

- I : b is present on $\mathcal{P}(r,v')$.
- II : b is **not** present on $\mathcal{P}(r,v')$.

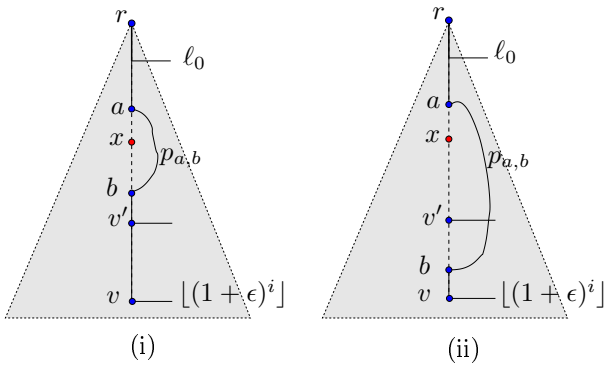

Figure 5: $p_{a,b}$ is the detour of $\mathcal{P}(r,v,x)$. (i) detour of type I, (ii) detour of type II

Let us consider the case when the detour $p_{a,b}$ is of type I. In this case, let w be the farthest ancestor of v such that $w \in S$ and the level of w is greater or equal to the level of b. Note that $p_{a,b}$ is also the detour of $\mathcal{P}(r,w,x)$, and so w would already have handled it in its data structure (this is because we process the special vertices in a top down fashion while building their data structures). Hence

PATH(w,x) would be storing $(1+2\epsilon')$-approximation of $\mathcal{P}(r,w,x)$. The structure of detour of type I can be exploited to make the following crucial observation.

Observation 4.2 *If* PATH(w,x) *is* $1+2\epsilon'$*-approximation of* $\mathcal{P}(r,w,x)$*, then* PATH$(w,x)\cdot\mathcal{P}(w,v)$ *will be* $(1+2\epsilon')$*-approximation of* $\mathcal{P}(r,v,x)$*.*

Using Observation 4.2, PATH(v,x) just stores a pointer to PATH(w,x) to handle this case (of detour I).

Let us now consider the case when the detour $p_{a,b}$ is of type II. Unfortunately, Observation 4.2 no longer holds in this case. So for vertex v, we cannot rely on its ancestors to take care of detour of type II. However, we can employ the following observation associated with the detours of type II.

Observation 4.3 *Let* $\alpha_1, \alpha_2, \cdots, \alpha_t$ *be the vertices on* $\mathcal{P}(r,v')$ *(in increasing level order) such that the detour of* $\mathcal{P}(r,v,\alpha_i)$ *is of Type II for all* i*. Then* $\delta(r,v,\alpha_1) \geq \delta(r,v,\alpha_2) \geq \cdots \geq \delta(r,v,\alpha_t)$*.*

It follows from Observation 4.3 that if $\delta(r,v,\alpha_i) \leq (1+\epsilon')\delta(r,v,\alpha_j)$ for any $i<j$, then $\mathcal{P}(r,v,\alpha_i)$ may as well serve as $(1+\epsilon')$-approximate shortest path from r to v avoiding α_j. In this situation, we need not store the path $\mathcal{P}(r,v,\alpha_j)$ if we are already storing $\mathcal{P}(r,v,\alpha_i)$. Using this observation, special vertex v will have to explicitly store only $O(\log_{1+\epsilon'} n)$ paths for all detours of type II. Moreover, we do not need to store explicitly those paths whose length is much larger than LEVEL(v). Specifically, if $\delta(r,v,x) \geq \frac{1}{\epsilon'}$LEVEL$(v)$, then it follows from Equation 1 that $\delta(r,\text{SUCC}(x),x)+\delta(\text{SUCC}(x),v)$ is $(1+2\epsilon')$-approximation of $\delta(r,v,x)$. Hence the data structure $\mathcal{H}_d$ itself takes care of such a case. This ensures that each path which v has to store explicitly will have $O(\frac{1}{\epsilon'}\text{LEVEL}(v))$ length.

Based on the detailed description of various cases as given above, Algorithm 2 presents the construction of the data structure associated with a special vertex v. The following lemma is a consequence

Algorithm 2: Computation of the data structure for a special vertex v

```
ℓ ← LEVEL(v); d ← ∞; P ← NULL;
foreach (vertex x ∈ P(r,v) in the increasing order of level) do
    if (δ(r,v,x) ≥ ℓ/ε') or (LEVEL(x) ≥ ℓ/(1+ε')) then
        PATH(v,x) keeps a pointer to P(r, SUCC(x), x) which is stored in H_d;
    else
        Let p_{a,b} be the detour of P(r,v,x);
        if p_{a,b} is of type I then
            Let w ∈ S be the farthest ancestor of v with LEVEL(w) ≥ LEVEL(b) ;
            PATH(v,x) stores pointer to PATH(w,x);
        else
            if d ≤ (1+ε')δ(r,v,x) then
                PATH(v,x) stores a pointer to P;
            else
                PATH(v,x) explicitly stores the entire path P(r,v,x);
                d ← δ(r,v,x);   P ← P(r,v,x);
```

of the discussion above.

Lemma 4.2 *Let* u *be a special vertex and* $x \in \mathcal{P}(r,u)$ *be such that* $u \in D_x$*. Then* PATH(u,x) *is an* $(1+2\epsilon')$*-approximation of* $\mathcal{P}(r,u,x)$*.*

4.3 Reporting $(1+\epsilon)$-approximate shortest paths from r using $\mathcal{H}_d^+$

Consider failure of any vertex $x \in V$. Let v be any vertex in D_x and $v \notin S$. Let u be the special vertex to which v is assigned, that is, $u = S(v)$. We can report approximate shortest path from r to v avoiding x as follows.

If x lies on $\mathcal{P}(u,v)$, we resort to the data structure $\mathcal{H}_d$ and report the path $\mathcal{P}(r,\text{SUCC}(x),x)\cdot\mathcal{P}(\text{SUCC}(x),v)$. Its length is bounded by $\delta(r,v,x)+2\delta(\text{SUCC}(x),v)$. Now observe that $\delta(\text{SUCC}(x),v) \leq$

$\delta(u,v)$, and it follows from Lemma 4.1 that $\delta(u,v)$ is at most $2\epsilon'\delta(r,v)$. Hence in this case, the reported path will have length at most $(1+4\epsilon')\delta(r,v,x)$.

If x does not lie on $\mathcal{P}(u,v)$, then we employ the data structure associated with special vertex u. We report $\text{PATH}(u,x)\cdot\mathcal{P}(u,v)$ as approximate path from r to v avoiding x. Length of this path can be bounded using Lemma 4.2 as follows.

$$\begin{aligned}(1+2\epsilon')\delta(r,u,x)+\delta(u,v) &\leq (1+2\epsilon')\delta(r,v,x)+2(1+\epsilon')\delta(u,v)\\ &\leq (1+2\epsilon')\delta(r,v,x)+4\epsilon'\delta(r,v) \quad \{\text{using Lemma 4.1 }\}\\ &\leq 1+6\epsilon')\delta(r,v,x)\end{aligned}$$

Thus setting $\epsilon'=\epsilon/6$ implies the following lemma.

Lemma 4.3 *For any failed vertex x and any vertex $v\in D_x$, the data structure $\mathcal{H}_d^+$ can report $(1+\epsilon)$-approximate shortest path from r to v avoiding x.*

4.4 Analysis of the data structure for $(1+\epsilon)$-approximate shortest paths

4.4.1 Space analysis

Recall that the data structure for singe source 3-approximate shortest paths avoiding any failed vertex requires $O(n\log n)$ space. The only extra space in the data structure for $(1+\epsilon)$-approximate shortest paths is due to the data structures associated with the set of special vertices. We can bound this extra space as follows. We need to analyze the space occupied by the data structures associated with all the special vertices. Let v be any special vertex. For each failed vertex $x\in\mathcal{P}(r,v)$, if $\delta(r,v,x)>\frac{\text{LEVEL}(v)}{\epsilon}$ or the detour associated with $\mathcal{P}(r,v,x)$ is of type I, $\text{PATH}(v,x)$ requires only $O(1)$ space. Thus the total space required for such vertices in the data structure of v is clearly $O(\text{LEVEL}(v))$. So let us consider the remaining vertices on $\mathcal{P}(r,v)$. Let y be one such vertex. The detour associated with the path $\mathcal{P}(r,v,y)$ must be of type II and $\delta(r,v,y)=O(\frac{\text{LEVEL}(v)}{\epsilon})$ must hold. It follows from Algorithm 2 that we shall store only $O(\log_{1+\epsilon}\frac{\text{LEVEL}(v)}{\epsilon})$ such paths explicitly. Furthermore, the sequence of lengths of these paths is a geometrically decreasing sequence with common ratio $(1+\epsilon)$. Hence the space required for storing all such paths in the data structure associated with v is $O(\text{LEVEL}(v)/\epsilon^2)$. So the overall space required by the data structure associated with v is $O(\text{LEVEL}(v)/\epsilon^2)$. Now it follows from Observation 4.1 that there are $\Omega(\epsilon\text{LEVEL}(v))$ descendants of v in T_r which are uniquely assigned to v. So all special vertices contribute a total of $O(n/\epsilon^3)$ space to the data structure. Hence we have proved the following Lemma.

Lemma 4.4 *The data structure for single source $(1+\epsilon)$-approximate shortest paths avoiding any failed vertex occupies $O(\frac{n}{\epsilon^3}+n\log n)$ space.*

4.4.2 Preprocessing time

Let us address the preprocessing time for computing the data structure associated with special vertices. For each special vertex v, we employ Algorithm 2 to compute the data structure associated with it. The entire running time of Algorithm 2 for a special vertex v is dominated by the computation of $\mathcal{P}(r,v,x)$ and $\delta(r,v,x)$ for each $x\in\mathcal{P}(r,v)$. We provide below a two-step algorithm to compute $\mathcal{P}(r,v,x)$ and $\delta(r,v,x)$ for each special vertex $v\in S$ and $x\in\mathcal{P}(r,v)$.

- For each special vertex v lying at level $\geq\sqrt{n/\epsilon}$ in T_r, we employ $O(m)$ time algorithm of Nardelli et al. [20] to compute $\delta(r,v,x)$ and $\mathcal{P}(r,v,x)$ for all $x\in\mathcal{P}(r,v)$. It follows from Observation 4.1 that there are at least $\epsilon\text{LEVEL}(v)$ descendants from T_r which are uniquely assigned to v. Therefore, the number of special vertices at level $\geq\sqrt{n/\epsilon}$ is not more than $O(\sqrt{n/\epsilon})$. So the total running time of this step is $O(m\sqrt{n/\epsilon})$.
- We execute the algorithm mentioned in Lemma 2.2 for handling failure of any vertex lying up to level $<\sqrt{n/\epsilon}$ in T_r. This would support efficient retrieval of $\delta(r,v,x)$ and $\mathcal{P}(r,v,x)$ for each special vertex v up to level $<\sqrt{n/\epsilon}$. The total running time of this step is $O(m\sqrt{n/\epsilon})$.

Thus the total preprocessing time of the data structures associated with all the special vertices is $O(m\sqrt{n/\epsilon})$. This bound along with Lemmas 4.3 and 4.4 lead to the following Theorem.

Theorem 4.1 *Given an undirected unweighted graph $G = (V, E)$, source $r \in V$, and any $\epsilon > 0$, we can build a data structure of size $O(n/\epsilon^3 + n \log n)$ in $O(m\sqrt{n/\epsilon})$ time which can report $(1+\epsilon)$-approximate shortest path from r to any $v \in V$ avoiding any single failed vertex $x \in V$ in time which is optimal up to a constant factor.*

4.5 A miscellaneous application

We would like to mention one application where our data structure for single source $(1+\epsilon)$-approximate shortest paths avoiding any failed vertex proves to be useful.

4.5.1 Nearest marked vertex problem under single vertex failure

Suppose there is a set $S \subset V$ of, so called, *marked* vertices in a given graph. Consider the problem of building a data structure which, for any $v, x \in V$, can report the vertex from S nearest to v when x has failed. We can use Theorem 4.1 to design a compact data structure for the approximate version of this problem. For any $v, x \in V$, this data structure will report a vertex $w \in S$ such that $\delta(v, w, x) \leq (1+\epsilon)\delta(v, S, x)$ in $O(1)$ time.

1. Add a dummy vertex r to the graph and join it to every vertex of set S. Let G' be the new graph thus formed.
2. With source vertex r and graph G', build the data structure for $(1+\epsilon)$-approximate shortest paths avoiding any failed vertex as mentioned in Theorem 4.1. We can easily augment this data structure suitably so that it takes constant time to report the neighbor of r on the $(1+\epsilon)$-approximate shortest path between r and v upon failure of any vertex x.

We can thus state the following theorem.

Theorem 4.2 *For any unweighted graph $G = (V, E)$ and a set $S \subseteq V$ of marked vertices, there exists a data structure of size $O(n/\epsilon^3 + n \log n)$ which can solve the approximate version of the nearest marked vertex problem under single vertex failure. The preprocessing time of the data structure is $O(m\sqrt{n/\epsilon})$ and the query time guaranteed is $O(1)$.*

5 All-pairs $(2k-1)(1+\epsilon)$-approximate shortest paths oracle avoiding a failed vertex

We start with an overview of the approximate distance oracle of Thorup and Zwick [24]. We then provide a brief description of our ideas which extend this oracle to handle any single vertex failure.

5.1 Overview of the approximate distance oracle of Thorup and Zwick [24]

The most impressive features of the $(2k-1)$-approximate distance oracle of Thorup and Zwick [24] are $O(k)$ query time and $O(kn^{1+1/k})$ size. Note that the size is subquadratic for any $k > 1$. To achieve such a compact size, the oracle stores distances from each vertex to only a *small* set of vertices that ensures the following key property. For every pair of vertices $u, v \in V$, there is some vertex w which is *near* to both u and v, and its distance to each of them is known. This property allows the oracle to report $\delta(u, w) + \delta(v, w)$ as an approximation for the distance $\delta(u, v)$.

The building block of the approximate distance oracle of Thorup and Zwick [24] is a novel structure called *Ball* which is defined as follows.

Definition 5.1 *Given a graph $G = (V, E)$, a vertex $v \in V$, and two subsets of vertices X and Y, the set $Ball(v, X, Y)$ consists of all those vertices of set X which lie within distance $\delta(v, Y)$ from v. In precise words,*

$$Ball(v, X, Y) = \{x \in X | \delta(v, x) < \delta(v, Y)\}$$

It is easy to observe that $Ball(v, X, Y) = \emptyset$ for any $v \in Y$ and $Ball(v, X, \emptyset) = X$. The following lemma shows that randomization (in construction of Y) can be used to achieve a small size of a ball. Its proof requires an elementary application of Chernoff bound.

Lemma 5.1 *[24] For a given subset $X \subseteq V$, let $Y \subset X$ be formed by selecting each vertex from X independently with probability $p > 0$. Then the size of $Ball(v, X, Y)$ is $O(1/p)$ in expectation and $O(\log n/p)$ with high probability, that is, with probability exceeding $1 - \frac{1}{n^c}$ for any positive constant c.*

We now provide an overview of the $(2k-1)$-approximate distance oracle of Thorup and Zwick [24]. It builds a $k+1$ level hierarchy $\mathcal{A}_k = \{A_0, A_1, ..., A_{k-1}\}$ of subsets of vertices defined as follows. $A_0 = V$, $A_k = \emptyset$, and A_i for any $0 < i < k$ is formed by selecting each vertex of A_{i-1} independently with probability $n^{-1/k}$. The oracle stores the following information for each vertex $v \in V$ and $i < k$.

- distance to each vertex of the set $Ball(v, A_i, A_{i+1})$ (this information is kept in a hash table).
- the vertex from A_i nearest to v (to be denoted as $p_i(v)$).

For a better illustration, see Figure 6 for the distance information stored at a vertex $v \in V$ in case of 3-approximate distance oracle, that is, $k = 2$.

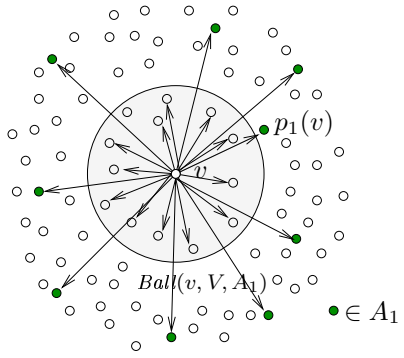

Figure 6: 3-approximate distance oracle : v stores distances to all the vertices pointed by arrows.

It follows from Lemma 5.1 that the space occupied by the entire data structure will be $O(kn^{1+1/k} \log n)$ with high probability. The data structure supports the following basic operation in $O(1)$ time.

Report distance between v and w if $w \in Ball(v, A_i, A_{i+1})$ for any given $v \in V$ and $w \in A_i \backslash A_{i+1}$.

In order to report approximate distance between any pair of vertices u and v, the oracle performs a series of such basic operations. At the end of $O(k)$ such operations, it succeeds in finding a vertex $w \in A_i \backslash A_{i+1}$ for some $i < k$ which has the following properties. w is present in $Ball(u, A_i, A_{i+1})$ as well as $Ball(v, A_i, A_{i+1})$, and either $\delta(u, w) \leq i\delta(u, v)$ or $\delta(v, w) \leq i\delta(u, v)$. The oracle finally reports $\delta(u, w) + \delta(v, w)$ as approximate distance between u and v. Using triangle inequality and the fact that $i < k$, the distance reported is at most $(2k-1)\delta(u, v)$.

We would also like to mention about one more novel structure defined by Thorup and Zwick [24]. This structure, called *cluster*, is basically *inverse* of a ball. For any sets $X \subseteq V$, $Y \subseteq X$ and any vertex $w \in X$, cluster $C(w, X, Y)$ is defined as

$$C(w, X, Y) = \{v \in V | \delta(v, w) < \delta(v, Y)\}$$

The following equality is an immediate consequence of the fact that balls and clusters are inverses of each others.

$$\sum_{v \in V} |Ball(v, X, Y)| = \sum_{w \in X} |C(w, X, Y)| \tag{2}$$

5.2 Overview of all-pairs approximate distance oracle avoiding any failed vertex

The basic structures and notations introduced by Thorup and Zwick [24] for the approximate distance oracle get extended in the case of single vertex failure quite naturally as follows.

- Balls and clusters in case of vertex failure are defined as:

$$Ball^x(v, A, B) = \{w \in A | \delta(v, w, x) < \delta(v, B, x)\}$$

$$C^x(w, A, B) = \{v \in V | \delta(v, w, x) < \delta(v, B, x)\}$$

- $p_i^x(v)$: the vertex from A_i which is nearest to v in $G \backslash \{x\}$.

Lemma 5.1 also gets extended easily as follows.

Lemma 5.2 *[24] For a given subset $X \subseteq V$, let $Y \subset X$ be formed by selecting each vertex from X independently with probability $p > 0$. Then, with high probability, the size of $Ball^x(v, X, Y)$ is $O(\log n/p)$ for each $x, v \in V$.*

Along the lines of the approximate distance oracle of Thorup and Zwick [24], the basic operation which the oracle avoiding any failed vertex should support is the following :

o : Report (exact or approximate) shortest path between v and w if $w \in Ball^x(v, A_i, A_{i+1})$ for any given $v, x \in V$ and $w \in A_i \backslash A_{i+1}$.

However, it can be observed that we will have to support this operation implicitly. This is because storing $Ball^x(v, A_i, A_{i+1})$ explicitly for all v, x, i can not be achieved in subquadratic space. To achieve this goal, our starting point is the simple observation that clusters and balls are inverses of each other. As a result we realize that $w \in Ball^x(v, A_i, A_{i+1})$ if and only if $v \in C^x(w, A_i, A_{i+1})$. Now we make use of the following insightful observation about the subgraph $\mathcal{G}_i(w)$ induced by the vertices of set $\cup_{x \in V} C^x(w, A_i, A_{i+1})$: This subgraph preserves the path $\mathcal{P}(w, v, x)$ for each $x, v \in V$ if $w \in Ball^x(v, A_i, A_{i+1})$. So we may build the data structure for single source approximate shortest paths avoiding vertex failure on graph $\mathcal{G}_i(w)$ with w as the source. Keeping this data structure for each $w \in A_i$ provides an implicit way for supporting the operation o. Using Theorem 4.1 and ignoring the logarithmic factors, it can be seen that the space required at a level i will be of the order of $\sum_{w \in A_i} |\cup_x C^x(w, A_i, A_{i+1})|$. However, it is not clear whether we can get an upper bound of the order of $n^{1+1/k}$ on this quantity. Here, as a new tool, we introduce the notion of ϵ-trimmed balls and clusters.

Definition 5.2 *Given a vertex x, any subsets A, B of vertices, and $\epsilon > 0$*

$$Ball^x(v, A, B, \epsilon) = \left\{ w \in A | \delta(v, w, x) < \frac{\delta(v, B, x)}{1 + \epsilon} \right\}$$

$$C^x(w, A, B, \epsilon) = \left\{ v \in V | \delta(v, w, x) < \frac{\delta(v, B, x)}{1 + \epsilon} \right\}$$

Instead of dealing with the usual balls (and clusters), we deal with ϵ-trimmed balls (and clusters). The key role played by ϵ-trimmed balls is that there exists a small set S of $O(\frac{\log n}{\epsilon})$ vertices such that

$$\cup_{x \in V} Ball^x(v, A_i, A_{i+1}, \epsilon) \subseteq \cup_{x \in S} Ball^x(v, A_i, A_{i+1}) \tag{3}$$

This equation and Lemma 5.2 provide a bound of $O(n^{1/k} \frac{\log^2 n}{\epsilon})$ on the size of $\cup_{x \in V} Ball^x(v, A_i, A_{i+1}, \epsilon)$ with high probability. Once again, we make use of the inverse relationship between clusters and balls. Note that Equation 2 gets extended seamlessly to ϵ-trimmed balls and clusters under single vertex failure as well. That is,

$$\sum_{w \in A_i} |\cup_x C^x(w, A_i, A_{i+1}, \epsilon)| = \sum_{v \in V} |\cup_x Ball^x(v, A_i, A_{i+1}, \epsilon)| \tag{4}$$

This leads to an upper bound of $O(n^{1+1/k}\frac{\log^2 n}{\epsilon})$ on $\sum_{w\in A_i} |\cup_x C^x(w, A_i, A_{i+1}, \epsilon)|$ with high probability. In this way the overall space required by the data structure turns out to be greater than that of the $(2k-1)$-approximate distance oracle of Thorup and Zwick [24] by polynomial in $1/\epsilon$ and $\log n$ only.

Having given an overview we now proceed to provide the complete details of the all-pairs approximate distance oracle avoiding any failed vertex. The following subsection describes the key role played by ϵ-trimmed balls and the subgraph $\mathcal{G}_i(w)$.

5.3 ϵ-trimmed balls and the subgraph $\mathcal{G}_i(w)$

We first state and prove an important lemma.

Lemma 5.3 *In a given graph $G = (V, E)$, let v be any vertex and let $u = p_{i+1}(v)$. Let x_1 and x_2 be any two vertices on the path $\mathcal{P}(v, u)$ from v to u with x_1 preceding x_2 and $\delta(v, A_{i+1}, x_1) \leq (1+\epsilon)\delta(v, A_{i+1}, x_2)$. Then*

$$Ball^{x_1}(v, A_i, A_{i+1}, \epsilon) \subseteq Ball(v, A_i, A_{i+1}) \cup Ball^{x_2}(v, A_i, A_{i+1})$$

Proof: To prove the lemma it suffices to prove the following equivalent statement. Let w be any vertex in A_i. If w does not belong to $Ball(v, A_i, A_{i+1}) \cup Ball^{x_2}(v, A_i, A_{i+1})$, then w does not belong to $Ball^{x_1}(v, A_i, A_{i+1}, \epsilon)$ as well. We prove this statement by analyzing the following two cases.

Case 1 : The vertex x_2 is present in $\mathcal{P}(v, w, x_1)$.

Since $w \notin Ball(v, A_i, A_{i+1})$, therefore, $\delta(v, w) \geq \delta(v, u)$. Hence, using triangle inequality, it follows that $\delta(v, x_2) + \delta(x_2, w) \geq \delta(v, w) \geq \delta(v, u) = \delta(v, x_2) + \delta(x_2, u)$. Hence $\delta(x_2, w) \geq \delta(x_2, u)$. Moreover, since x_1 precedes x_2 on the path $\mathcal{P}(v, u)$, so x_1 does not appear on $\mathcal{P}(x_2, u)$, and so $\delta(x_2, u, x_1) = \delta(x_2, u)$. Hence

$$\delta(x_2, w, x_1) \geq \delta(x_2, u, x_1) \tag{5}$$

Now it is given that $x_2 \in \mathcal{P}(v, w, x_1)$, so using optimal subpath property it follows that

$$\begin{aligned} \delta(v, w, x_1) &= \delta(v, x_2, x_1) + \delta(x_2, w, x_1) \\ &\geq \delta(v, x_2, x_1) + \delta(x_2, u, x_1) \quad \{\text{using Equation 5}\} \\ &\geq \delta(v, u, x_1) \geq \delta(v, A_{i+1}, x_1) \end{aligned}$$

Hence it follows that w does not even belong to $Ball^{x_1}(v, A_i, A_{i+1})$. So, w won't belong to $Ball^{x_1}(v, A_i, A_{i+1}, \epsilon)$ as well since the latter is a subset of the former.

Case 2 : The vertex x_2 is not present in $\mathcal{P}(v, w, x_1)$.

In this case, we proceed as follows.

$$\begin{aligned} \delta(v, w, x_1) &= \delta(v, w, \{x_1, x_2\}) \geq \delta(v, w, x_2) \\ &\geq \delta(v, A_{i+1}, x_2) \quad \{\text{since } w \notin Ball^{x_2}(v, A_i, A_{i+1})\} \\ &\geq \frac{\delta(v, A_{i+1}, x_1)}{1+\epsilon} \end{aligned}$$

Hence it follows that $w \notin Ball^{x_1}(v, A_i, A_{i+1}, \epsilon)$. •

Now we shall use Lemma 5.3 to establish an upper bound on the size of set $\cup_{x\in V} Ball^x(v, A_i, A_{i+1}, \epsilon)$. Let $u = p_{i+1}(v)$ and let the shortest path $\mathcal{P}(u, v)$ be $\langle v(= x_0), x_1, ..., x_\ell(= u)\rangle$. It is easy to observe that $\cup_{x\in V} Ball^x(v, A_i, A_{i+1}, \epsilon) = \cup_{1\leq j\leq \ell} Ball^{x_j}(v, A_i, A_{i+1}, \epsilon)$. We shall show that there is a sequence α of $O(\frac{\log n}{\epsilon})$ monotonically increasing integers from the interval $[1, \ell]$ such that for each i, we have

$$\cup_{\alpha(i-1)<j\leq\alpha(i)} Ball^{x_j}(v, A_i, A_{i+1}, \epsilon) \subseteq Ball^{x_{\alpha(i)}}(v, A_i, A_{i+1}) \cup Ball(v, A_i, A_{i+1})$$

Now we describe an algorithm to construct the sequence α. For each vertex $x \in \mathcal{P}(v, u)$ (including vertex u), we define $value(x)$ as $\delta(v, A_{i+1}, x)$. Let h be the maximum $value$ of any node on the path $\mathcal{P}(v, u)$.

We define $h_1 = h$. We define $\alpha(1)$ to be the largest integer in the interval $[1, \ell]$ such that $value(x_{\alpha(1)}) \geq h_1/(1+\epsilon)$. It can be seen that for all $1 \leq j \leq \alpha(1)$, $\delta(v, A_{i+1}, x_j) \leq (1+\epsilon)\delta(v, A_{i+1}, x_{\alpha(1)})$. Therefore, it follows from Lemma 5.3 that for each vertex $x \in \{x_1, ..., x_{\alpha(1)}\}$, $Ball^x(v, A_i, A_{i+1}, \epsilon) \subseteq Ball^{x_{\alpha(1)}}(v, A_i, A_{i+1}) \cup Ball(v, A_i, A_{i+1})$. Hence

$$\cup_{0<j\leq\alpha(1)} Ball^{x_j}(v, A_i, A_{i+1}, \epsilon) \subseteq Ball^{x_{\alpha(1)}}(v, A_i, A_{i+1}) \cup Ball(v, A_i, A_{i+1})$$

We define $h_2 = \max\{value(x_j) | \alpha(1) < j \leq \ell\}$. It follows from the construction that $h_2 < h/(1+\epsilon)$, We define $\alpha(2)$ to be the largest integer in the interval $[\alpha(1)+1, ..., \ell]$ such that $value(x_{\alpha(2)}) \geq h_2/(1+\epsilon)$. It can be seen that for all $\alpha(1) < j \leq \alpha(2)$, $\delta(v, A_{i+1}, x_j) \leq (1+\epsilon)\delta(v, A_{i+1}, x_{\alpha(2)})$. Therefore, it follows from Lemma 5.3 that for each vertex $x \in \{x_{\alpha(1)+1}, ..., x_{\alpha(2)}\}$, $Ball^x(v, A_i, A_{i+1}, \epsilon) \subseteq Ball^{x_{\alpha(2)}}(v, A_i, A_{i+1}) \cup Ball(v, A_i, A_{i+1})$. Hence

$$\cup_{\alpha(1)<j\leq\alpha(2)} Ball^{x_j}(v, A_i, A_{i+1}, \epsilon) \subseteq Ball^{x_{\alpha(2)}}(v, A_i, A_{i+1}) \cup Ball(v, A_i, A_{i+1})$$

We define $h_3 = \max\{value(x_j) | \alpha(2) < j \leq \ell\}$. It follows from the construction that $h_3 < h_2/(1+\epsilon) < h/(1+\epsilon)^2$, We define $\alpha(3)$ to be the largest integer in the interval $[\alpha(2)+1, ..., \ell]$ such that $value(x_{\alpha(3)}) \geq h_3/(1+\epsilon)$. In this manner, we continue scanning the path $\mathcal{P}(v, u)$ from v to u to compute the elements of sequence α. The last element to be added to this sequence will be ℓ. Note that $\langle h_i \rangle$ is a geometrically decreasing sequence with common ratio $(1+\epsilon)$. So the number of elements in the sequence will be of the order of $\log_{1+\epsilon} h = O(\frac{\log h}{\epsilon})$. Our desired set S is defined as $\{x_j | j \in \alpha\}$. Note that $u \in S$ and also observe that $Ball(v, A_i, A_{i+1}) \subseteq Ball^u(v, A_i, A_{i+1})$ since $u \in A_{i+1}$. We can thus conclude that there is a set S of $O(\frac{\log n}{\epsilon})$ vertices such that

$$\cup_{x\in V} Ball^x(v, A_i, A_{i+1}, \epsilon) \subseteq \cup_{x\in S} Ball^x(v, A_i, A_{i+1})$$

Using Lemma 5.2 and the above equation, we can conclude the following Theorem.

Theorem 5.1 *Let $G = (V, E)$ be an unweighted graph and $\mathcal{A}_k$ be the hierarchy of vertices as defined earlier. For any vertex v, integer $i < k-1$, and constant $\epsilon > 0$, with very high probability*

$$|\cup_{x\in V} Ball^x(v, A_i, A_{i+1}, \epsilon)| = O\left(n^{1/k} \frac{\log^2 n}{\epsilon}\right)$$

Now recall Equation 4 which states that balls and clusters are inverses of each others, even under vertex failure. This equation and Theorem 5.1 imply the following corollary.

Corollary 5.1 *Let $G = (V, E)$ be an undirected unweighted graph and $\mathcal{A}_k$ be the hierarchy of vertices as defined earlier. For any integer $i < k-1$, and constant $\epsilon > 0$, with high probability*

$$\sum_{w\in A_i} |\cup_{x\in V} C^x(w, A_i, A_{i+1}, \epsilon)| = O\left(n^{1+1/k} \frac{\log^2 n}{\epsilon}\right)$$

Now we describe the key role played by the graph $\mathcal{G}_i(w)$. Recall that $\mathcal{G}_i(w)$ is the subgraph of the original graph induced by vertices of the set $\cup_{x\in V} C^x(w, A_i, A_{i+1}, \epsilon)$. The following lemma highlights an important fact about $\mathcal{G}_i(w)$.

Lemma 5.4 *If $w \in Ball^x(v, A_i, A_{i+1}, \epsilon)$, then the shortest path $\mathcal{P}(w, v, x)$ is present in the subgraph $\mathcal{G}_i(w) \backslash \{x\}$.*

Proof: Let y be any vertex on the shortest path $\mathcal{P}(w, v, x)$. We shall first prove that w belongs to $Ball^x(y, A_i, A_{i+1}, \epsilon)$. The proof is based on contradiction. Let $w \notin Ball^x(y, A_i, A_{i+1}, \epsilon)$. So there must be a vertex $z \in A_{i+1}$ such that

$$\delta(y, z, x) \leq (1+\epsilon)\delta(y, w, x)$$

Since the vertex y belongs to the shortest path between w and v in $G\backslash\{x\}$, therefore, the following inequality can be concluded.

$$\delta(v,z,x) \leq (1+\epsilon)\delta(v,w,x)$$

However, it implies that $w \notin Ball^x(v, A_i, A_{i+1}, \epsilon)$, a contradiction.

So for each vertex y on the shortest path $\mathcal{P}(w,v,x)$, w belongs to $Ball^x(y, A_i, A_{i+1}, \epsilon)$. Hence $y \in C^x(w, A_i, A_{i+1}, \epsilon)$, and so the entire path $\mathcal{P}(w,v,x)$ is present in $\mathcal{G}_i(w)$. Moreover, since $x \notin \mathcal{P}(w,v,x)$, the shortest path $\mathcal{P}(w,v,x)$ is present in $\mathcal{G}_i(w)\backslash\{x\}$. •

So if $w \in Ball^x(v, A_i, A_{i+1}, \epsilon)$, then $(1+\epsilon)$-approximate distance between v and w avoiding x can be reported using the data structure for single source $(1+\epsilon)$-approximate shortest paths from w avoiding any failed vertex in the graph $\mathcal{G}_i(w)$ (see Theorem 4.1).

5.4 The data structures

The $(2k-1)(1+\epsilon)$-approximate distance oracle avoiding any failed vertex will keep the following data structures for each $i < k$.

- Let $\mathcal{N}_i$ be the data structure for the *nearest marked vertex* problem with $S = A_i$ as described in Theorem 4.2. Recall that for any $v, x \in V$, this data structure reports a vertex $w \in A_i$ such that $\delta(v,w,x) \leq (1+\epsilon)\delta(v, A_i, x)$. Henceforth, we shall use $p_i^x(v,\epsilon)$ to denote this vertex as reported by $\mathcal{N}_i$.
- For each $w \in A_i$, let $\mathcal{D}_i(w)$ be the data structure for $(1+\epsilon)$-approximate shortest paths from w avoiding any failed vertex in the graph $\mathcal{G}_i(w)$.

It follows from Theorem 4.2 that the total space required by all $\mathcal{N}_i$'s will be $O(nk(\log n + \frac{1}{\epsilon^3})) = O(nk\frac{\log n}{\epsilon^3})$. It follows from Theorem 4.1 that the space required by $\mathcal{D}_i(w)$ will be of the order of $|\cup_x C^x(w, A_i, A_{i+1})| \cdot \frac{\log n}{\epsilon^3}$. Therefore, using Corollary 5.1, the total space required by $\mathcal{D}_i(w)$ for all $w \in A_i$ will be $O(\frac{1}{\epsilon^4} n^{1+1/k} \log^3 n)$ with high probability. So the total space occupied by the approximate distance oracle avoiding any failed vertex will be $O(\frac{k}{\epsilon^4} n^{1+1/k} \log^3 n)$ with high probability. We may rebuild the oracle if the size exceeds this bound by a factor of 2. This will require only $O(1)$ rebuildings in expectation. Hence we can state the following lemma.

Lemma 5.5 *The total space occupied by the approximate distance oracle avoiding any failed vertex is* $O(\frac{k}{\epsilon^4} n^{1+1/k} \log^3 n)$.

Now we describe the algorithm to retrieve approximate distance between any two vertices u and v upon failure of any vertex x. First we define a notation $\hat{\delta}(u,w,x)$ for any $w \in A_i$ as follows. If $w = p_i^x(u,\epsilon)$, then $\hat{\delta}(u,w,x)$ represents the approximate distance between u and w upon failure of x as reported by $\mathcal{N}_i$. Otherwise, $\hat{\delta}(u,w,x)$ denotes the approximate distance between u and w upon failure of x as reported by $\mathcal{D}_i(w)$. We now state the following simple observations.

Observation 5.1 *For any vertex $u \in V$ and a vertex $w \in A_i$, if $w = p_i^x(u,\epsilon)$ then $\hat{\delta}(u,w,x) \leq (1+\epsilon)\delta(u, A_i, x)$*

Observation 5.2 *If $w \in Ball^x(v, A_i, A_{i+1}, \epsilon)$, then $\hat{\delta}(v,w,x) \leq (1+\epsilon)\delta(v,w,x)$.*

We describe our query answering procedure in Algorithm 3. In this algorithm we assume that $\hat{\delta}(v,y,x) = \infty$ if $v \notin \mathcal{G}_i(y)$ and $\hat{\delta}(u,z,x) = \infty$ if $u \notin \mathcal{G}_i(z)$.

Let us now analyze the query answering algorithm to bound the stretch $\frac{d(u,v,x)}{\delta(u,v,x)}$. To retrieve approximate distance between u and v upon failure of vertex x, the aim is to find a vertex w which is *close* to both u and v, and its approximate distance to both u and v is known. The query answering algorithm confines the search for such vertices to the set $\{p_i^x(u,\epsilon) | i < k\} \cup \{p_i^x(v,\epsilon) | i < k\}$. The following lemma plays the key role in the analysis.

Algorithm 3: Retrieving approximate distance between u and v upon failure of vertex x

$d(u,v,x) \longleftarrow \infty$;
$i \leftarrow 0$;
foreach $i < k$ **do**
  Compute $y \leftarrow p_i^x(u,\epsilon)$ and $z \leftarrow p_i^x(v,\epsilon)$ using $\mathcal{N}_i$;
  $d(u,v,x) \leftarrow \min\left(d(u,v,x), \hat{\delta}(u,y,x) + \hat{\delta}(v,y,x), \hat{\delta}(u,z,x) + \hat{\delta}(v,z,x)\right)$;
return $d(u,v,x)$;

Lemma 5.6 *Let $G = (V,E)$ be an undirected unweighted graph, with $u, v \in V$ and j being any positive integer $< k$. If for each $i < j$, neither $p_i^x(u,\epsilon) \in Ball^x(v, A_i, A_{i+1}, \epsilon)$ nor $p_i^x(v,\epsilon) \in Ball^x(u, A_i, A_{i+1}, \epsilon)$, then*

$$\delta(u, p_j^x(u,\epsilon), x) \leq (1+\epsilon)^{2j} j\delta(u,v,x) \quad \textit{as well as} \quad \delta(v, p_j^x(v,\epsilon), x) \leq (1+\epsilon)^{2j} j\delta(u,v,x)$$

Proof: We provide a proof by induction on j.
Base Case : $j = 1$
Note that $p_0^x(u,\epsilon) = u$ and $p_0^x(v,\epsilon) = v$. If $u \notin Ball^x(v, A_0, A_1, \epsilon)$, then it must be that $\delta(v, A_1, x) \leq (1+\epsilon)\delta(u,v,x)$. Hence, by definition of $p_1^x(v,\epsilon)$,

$$\delta(v, p_1^x(v,\epsilon), x) \leq (1+\epsilon)^2 \delta(u,v,x)$$

Along similar lines, we can prove that $\delta(u, p_1^x(u,\epsilon), x) \leq (1+\epsilon)^2\delta(u,v,x)$. Hence the assertion holds for the base case.
Induction step :
Suppose the assertion holds for $j = t-1$. We shall prove the assertion for $j = t$. So we are given that for each $i < t$, neither $p_i^x(u,\epsilon) \in Ball^x(v, A_i, A_{i+1}, \epsilon)$ nor $p_i^x(v,\epsilon) \in Ball^x(u, A_i, A_{i+1}, \epsilon)$. Firstly, it follows from the induction hypothesis that

$$\delta(u, p_{t-1}^x(u,\epsilon), x) \leq (1+\epsilon)^{2(t-1)}(t-1)\delta(u,v,x) \tag{6}$$

$$\delta(v, p_{t-1}^x(v,\epsilon), x) \leq (1+\epsilon)^{2(t-1)}(t-1)\delta(u,v,x) \tag{7}$$

Now consider the vertex $p_{t-1}^x(u,\epsilon)$. Note that $p_{t-1}^x(u,\epsilon)$ belongs to A_{t-1}. Since it is given that $p_{t-1}^x(u,\epsilon) \notin Ball^x(v, A_{t-1}, A_t, \epsilon)$, therefore

$$\delta(v, A_t, x) \leq (1+\epsilon)\delta(v, p_{t-1}^x(u,\epsilon), x)$$

Hence

$$\delta(v, p_t^x(v,\epsilon), x) \leq (1+\epsilon)^2 \delta(v, p_{t-1}^x(u,\epsilon), x)$$

Now using triangle inequality

$$\delta(v, p_{t-1}^x(u,\epsilon), x) \leq \delta(u, p_{t-1}^x(u,\epsilon), x) + \delta(u,v,x)$$

Combining the above two inequalities, we get

$$\begin{aligned}
\delta(v, p_t^x(v,\epsilon), x) &\leq (1+\epsilon)^2 \left(\delta(u, p_{t-1}^x(u,\epsilon), x) + \delta(u,v,x)\right) \\
&\leq (1+\epsilon)^2 \left((1+\epsilon)^{2(t-1)}(t-1)\delta(u,v,x) + \delta(u,v,x)\right) \quad \{\text{using Equation 6}\} \\
&\leq (1+\epsilon)^{2t} t\delta(u,v,x)
\end{aligned}$$

Along similar lines we can prove that $\delta(u, p_t^x(u,\epsilon), x) \leq (1+\epsilon)^{2t} t\delta(u,v,x)$. This concludes the proof for $j = t$. Hence by principle of mathematical induction, the assertion holds for all j. •

We finally prove a bound on $d(u,v,x)$ using Lemma 5.6.

Lemma 5.7 *Let $d(v,u,x)$ be the approximate distance between u and v avoiding x as output by our query answering algorithm. Then, $d(u,v,x) \leq (1+\epsilon)^{2k-1}(2k-1)\delta(u,v,x)$.*

Proof: Let j be the smallest positive integer for which either $p_j^x(u,\epsilon) \in Ball^x(v,A_j,A_{j+1},\epsilon)$ or $p_j^x(v,\epsilon) \in Ball^x(u,A_j,A_{j+1},\epsilon)$ holds true. Since $Ball^x(v,A_{k-1},A_k,\epsilon) = A_{k-1}$, so $p_{k-1}^x(u,\epsilon) \in Ball^x(v,A_{k-1},A_k,\epsilon)$ as well as $p_{k-1}^x(v,\epsilon) \in Ball^x(u,A_{k-1},A_k,\epsilon)$. Hence $j \leq k-1$ always. Now without loss of generality, let $p_j^x(u,\epsilon) \in Ball^x(v,A_j,A_{j+1},\epsilon)$. So, it follows from Lemma 5.6 that

$$\delta(u,p_j^x(u,\epsilon),x) \leq (1+\epsilon)^{2j} j\delta(u,v,x) \tag{8}$$

Using triangle inequality, it follows that

$$\delta(v,p_j^x(u,\epsilon),x) \leq (1+\epsilon)^{2j}(j+1)\delta(u,v,x) \tag{9}$$

Furthermore, it follows from Observations 5.1 and 5.2 respectively that

$$\hat{\delta}(u,p_j^x(u,\epsilon),x) \leq (1+\epsilon)\delta(u,p_j^x(u,\epsilon),x) \quad \text{and} \quad \hat{\delta}(v,p_j^x(u,\epsilon),x) \leq (1+\epsilon)\delta(v,p_j^x(u,\epsilon),x)$$

Therefore, at the end of jth iteration, $d(u,v,x)$ can be bounded as follows.

$$\begin{aligned} d(u,v,x) &\leq \hat{\delta}(u,p_j^x(u,\epsilon),x) + \hat{\delta}(v,p_j^x(u,\epsilon),x) \\ &\leq (1+\epsilon)\left(\delta(u,p_j^x(u,\epsilon),x) + \delta(v,p_j^x(u,\epsilon),x)\right) \\ &\leq (1+\epsilon)^{2j+1}(2j+1)\delta(u,v,x) \quad \{\text{using Equations 8 and 9}\} \\ &\leq (1+\epsilon)^{2k-1}(2k-1)\delta(u,v,x) \quad \{\text{since } j \leq k-1\} \end{aligned}$$

This completes the proof. •

For bounding $(1+\epsilon)^{2k-1}$ below $(1+\epsilon')$ for a given ϵ', we may select sufficiently small value of ϵ (let $\epsilon = O(\epsilon'/k)$). So combining Lemma 5.5 and Lemma 5.7, we can thus conclude with the following theorem.

Theorem 5.2 *Given an integer $k > 1$ and a fraction $\epsilon > 0$, an unweighted graph $G = (V,E)$ can be processed to construct a data structure which can answer $(2k-1)(1+\epsilon)$-approximate distance query between any two nodes $u \in V$ and $v \in V$ upon failure of any vertex $x \in V$ in $O(k)$ time. The total size of the data structure is $O(\frac{k^5}{\epsilon^4} n^{1+1/k} \log^3 n)$.*

We would like to state that the data structure mentioned in Theorem 5.2 can be preprocessed in $O(kmn^{1+1/k})$ time. The preprocessing algorithm is quite straightforward and employs the $O(kmn^{1/k})$ time algorithm of approximate distance oracles of Thorup and Zwick [24].

6 Conclusions

In this paper, we presented compact data structures for approximate shortest paths avoiding any failed vertex in undirected unweighted graphs. The size of these data structures is nearly optimal and the query time guaranteed is optimal up to a constant factor. We presented almost linear (in n) size data structures for single source approximate shortest paths avoiding any failed vertex. We also presented an all-pairs approximate shortest paths oracle avoiding any failed vertex in undirected unweighted graphs. This oracle is obtained by suitable adaptation of the approximate distance oracle of Thorup and Zwick [24] using clever insights and new ideas. Interestingly, the size stretch trade-off of the oracle remains nearly preserved though the oracle is now tolerant to any single vertex failure. We would like to conclude with the following open problems for future research.

- *Single source $(1+\epsilon)$-approximate shortest paths in a weighted graph*:
  Can we design a data structure for undirected weighted graphs which occupies $O(n \operatorname{polylog} n)$ space and can report $(1+\epsilon)$-approximate shortest paths from a designated source avoiding any failed vertex for any $\epsilon > 0$?

- *Data structures for distributed environment:*
  The data structure for the single source approximate shortest paths avoiding any failed vertex, as presented in this article, is for centralized environment only. Note that there are plenty of distributed settings in networks, where one is interested in traveling from a source node to any given destination, avoiding a reported failure (see [23]). It would be interesting and useful to adapt our centralized data structure in the distributed environment.

- *All-pairs approximate distance oracle for weighted graphs:*
  Our data structure for the single source 3-approximate shortest paths avoiding any failed vertex in weighted graphs (Theorem 3.2) can be used to build all-pairs approximate distance oracles capable of tolerating any single vertex failure for weighted graphs. Though the space occupied will be $O(n^{1+1/k} \operatorname{polylog} n)$, the stretch would be exponential in k. Therefore, it would be an interesting problem in the domain of weighted graphs to design all-pairs approximate distance oracles capable of tolerating any single vertex failure which occupy $O(n^{1+1/k} \operatorname{polylog} n)$ space and still guarantee $O(k)$ stretch.

- *Multiple vertex failures:*
  Can we design approximate shortest paths oracles which may handle failure of two or more vertices ? For the all-pairs exact distances, Duan and Pettie [11] presented a distance sensitivity oracle which occupies $O(n^2 \log^3 n)$ size and can handle failures of any two vertices at a time.

Acknowledgments

Part of the work was done while the authors were at Max-Planck Institute for Computer Science, Saarbruecken, Germany for the period May-July, 2009. The authors are also grateful to anonymous referees for providing useful comments which led to improving the readability of the paper.

References

[1] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28(4):1167–1181, 1999.

[2] S. Baswana and T. Kavitha. Faster algorithms for all-pairs approximate shortest paths in undirected graphs. *SIAM Journal on Computing*, 39(7):2865–2896, 2010.

[3] S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in expected $O(n^2)$ time. *ACM Transactions on Algorithms*, 2(4):557–577, 2006.

[4] A. Bernstein. A nearly optimal algorithm for approximating replacement paths and k shortest simple paths in general graphs. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 742–755, 2010.

[5] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 101–110, 2009.

[6] S. Chechik, M. Langberg, D. Peleg, and L. Roditty. f-sensitivity distance oracles and routing schemes. In *Proceedings of the 18th Annual European Symposium on Algorithms*, pages 84–96, 2010.

[7] E. Cohen and U. Zwick. All-pairs small-stretch paths. *Journal of Algorithms*, 38(2):335–353, 2001.

[8] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the ACM*, 51(6):968–992, 2004.

[9] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM Journal on Computing*, 37(5):1299–1318, 2008.

[10] D. Dor, S. Halperin, and U. Zwick. All-pairs almost shortest paths. *SIAM Journal on Computing*, 29(5):1740–1759, 2000.

[11] R. Duan and S. Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 506–515, 2009.

[12] P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice,1963)*, pages 29–36, Publ. House Czechoslovak Acad. Sci., Prague, 1964.

[13] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization problem. *Journal of the ACM*, 34:596–615, 1987.

[14] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[15] J. Hershberger and S. Suri. Vickrey prices and shortest paths: What is an edge worth? In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*, pages 252–259, 2001.

[16] J. Hershberger, S. Suri, and A. Bhosle. On the difficulty of some shortest path problems. *ACM Transactions on Algorithms*, 3:123–139, 2007.

[17] D. R. Karger, D. Koller, and S. J. Philips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22:1199–1217, 1993.

[18] E. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the k shortest paths problem. *Management Science*, 18:401–405, 1971/72.

[19] K. Malik, A. K. Mittal, and S. K. Gupta. The k most vital arcs in the shortest path problem. *Operation Research Letters*, 4:223–227, 1989.

[20] E. Nardelli, G. Proietti, and P. Widmayer. Finding the most vital node of a shortest path. *Theoretical Computer Science*, 296(1):167–177, 2003.

[21] L. Roditty and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. In *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming*, pages 249–260, 2005.

[22] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[23] M. Thorup. Fortifying OSPF/IS-IS against link-failure. manuscript, 2001.

[24] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[25] J. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17:712–716, 1970/71.