Compact and Efficient Fault Tolerant Structures for Directed Graphs

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Keerti Choudhary

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

August, 2017

Synopsis

Single source reachability, shortest paths, strong connectivity, etc. are some of the most fundamental problems in graph algorithms. In the static setting each of these problems can be solved in $\Theta(n + m)$ time, where n and m are respectively the number of vertices and edges in the graph. However, most of the applications in real life deal with graphs which are prone to failures. These failures are both small in number and transient in nature. This aspect is generally captured by associating a parameter k with the network such that there are at most k vertices (or edges) that are failed at any stage. Here k is much smaller than the number of vertices in the underlying graph. Our main goal is to compute data structures that can achieve robustness by being functional even after the occurrence of failures. The three main objectives studied in this scenario are: (i) computing a compact data structure (*oracle*) that can quickly answer any query for a given problem (e.g. distance, connectivity) on occurrence of any k failures, (ii) designing a compact *labeling scheme* and *routing scheme* in network avoiding the failed nodes/edges, and (iii) computing a *sparse subgraph* that preserves a certain property (e.g. shortest path, connectedness, etc.) of the graph even after failures have occurred. In this thesis, we address one or more of these objectives for some of the fundamental problems in the fault tolerant model for directed graphs.

We first consider the problem of computing a sparse subgraph that preserves the reachability from a designated source even after the failure of any k edges or vertices. We settle this problem for any $k > 1$, by showing an upper bound of $2^k n$. We also show that this bound is tight by proving a lower bound of $\Omega(2^k n)$. For the problem of dominators, we present an alternative construction that uses $O(m \log n)$ time and space. Next, we obtain a fault tolerant oracle for answering reachability queries from a designated source on failure of any two vertices in constant time. For the strong connectivity problem, we obtain an oracle that after any k failures can report all the strongly connected components of the remaining graph in $O(2^k n \log^2 n)$ time. Our

reporting time is optimal (up to logarithmic factors) for any fixed value of k. We also address the problem of maintaining $(1+\epsilon)$-approximate shortest paths in a directed weighted graph from a designated source in the presence of a failure of an edge or a vertex. We obtain near optimal results for an oracle, a subgraph, a labeling scheme, and a routing scheme for this problem. Also we show that the space used by our oracle and subgraph is optimal up to logarithmic factors by providing a matching lower bound.

Acknowledgments

I feel extremely grateful and indebted to my advisor, Prof. Surender Baswana for his immense help throughout the journey of my Ph.D. Without his guidance, motivating ideas, and insightful discussions this thesis could not have been possible. I feel privileged to be his student and am thankful to him for introducing me to the wonderful world of randomization and data structures. From him I gained not only the techniques and thinking strategies, but also the art of perseverance and dedication. He was always very generous with his time, and encouraged and supported me whenever I needed. I am also thankful to Deepshikha Ma'am for her friendliness and utmost caring attitude. I really cherish the moments spent with Ma'am during my Ph.D and would always remember the tasty cakes she baked :).

I am thankful to Prof. Liam Roditty for his collaborations and hosting me multiple times in Israel. My visits to Bar-Ilan University were very fruitful. I also thank him for suggesting interesting problems to work on. It was a great learning experience working with him.

I want to express my sincere gratitude to my friends Anusha, Arzoo and Venkata as I could always count on them, and they have been motivating, advising and helping since my undergrad days. I am specially thankful to my Ph.D. friends Shubhangi, Hrishikesh and Garima who made me feel at-home on campus. I would also like thank Shuvasri, Annwesha, Shahbaz, Diptarko, Sumanto, Sumit, Ram, Debarati, Amit, and Rajendra for the great moments we had together.

Most importantly, I thank my parents, my brother and my sister for having a lot of hope in me, and for their advices in all stages of my life. My parents have always been very passionate about academics. Their engaging discussions in science and puzzles during my childhood days developed great curiosity in me towards mathematics and computer science. I feel grateful towards my family for their unconditional love and care.

Last but not the least, I thank the entire Computer science department at IITK and the Google

fellowship program for supporting me financially during my Ph.D.

Contents

List of Publications

[BCR15] **Fault Tolerant Reachability for Directed Graphs**

with Surender Baswana and Liam Roditty

International Symposium on Distributed Computing (DISC), 2015

[Cho16] **An Optimal Dual Fault Tolerant Reachability Oracle**

International Colloquium on Automata, Languages, and Programming (ICALP), 2016

[BCR16] **Fault tolerant subgraph for single source reachability: Generic and optimal**

with Surender Baswana and Liam Roditty

Symposium on the Theory of Computing (STOC), 2016

[BCR17] **An efficient strongly connected components algorithm in the fault tolerant model**

with Surender Baswana and Liam Roditty

International Colloquium on Automata, Languages, and Programming (ICALP), 2017

[BCHR18] **On single source fault tolerant approximate shortest path**

with Surender Baswana, Moazzam Hussain and Liam Roditty

Symposium on Discrete Algorithms (SODA), 2018

Chapter 2 is based on [BCR15], Chapter 3 is based on [Cho16], Chapter 4 is based on [BCR16], Chapter 5 is based on [BCR17], and Chapter 6 is based on [BCHR18].

List of Figures

List of Common Notations

$G = (V, E)$	A directed graph on $n = \|V\|$ vertices and $m = \|E\|$ edges
s	A designated source vertex in V
T	A reachability tree rooted at s
$parent_T(v)$	The parent of vertex v in tree T
$depth_T(v)$	The depth of vertex v in tree T
$T(v)$	The subtree of T rooted at a vertex v
$\text{PATH}_T(x, y)$	The path from vertex x to vertex y in tree T
$\text{PATH}_T(\bar{x}, y)$	$\text{PATH}_T(x, y) \setminus \{x\}$
$\text{PATH}_T(x, \bar{y})$	$\text{PATH}_T(x, y) \setminus \{y\}$
$\text{PATH}_T(\bar{x}, \bar{y})$	$\text{PATH}_T(x, y) \setminus \{x, y\}$
$\text{LCA}_T(u, v)$	The Lowest Common Ancestor of u and v in tree T
$\text{IDOM}(v)$	The immediate-dominator of vertex v in G
$P[x, y]$	The subpath of path P lying between vertices x, y, assuming x precedes y on P
$P[\cdot, y]$	The prefix of path P up to vertex y, assuming that y lies in P
$P::Q$	The path formed by concatenating paths P and Q in G, here it is assumed that the last vertex of P is the same as the first vertex of Q
$\text{OUT}(A)$	The set of all those vertices in $V \setminus A$ which have an incoming edge from some vertex in A, where $A \subseteq V$
$deg(A)$	The sum of the degrees of all vertices in the set A
$H(A)$	The subgraph of a graph H induced by the vertices of A
$H \setminus F$	The graph obtained by deleting the edges lying in a set F from graph H
$H + (u, v)$	The graph obtained by adding an edge (u, v) to graph H
$\text{IN-EDGES}(v, H)$	The set of all incoming edges to v in graph H
$dist_H(u, v)$	The distance of vertex v from vertex u in graph H

Chapter 1

Introduction

1.1 Problem Statement

There are many problems in theoretical as well as applied computer science which are modeled in terms of graphs. Some prominent examples include road networks, internet packet routing, social networks, traffic scheduling, broadcasting, etc. So it is important to design efficient algorithms for these graph problems. While we do have efficient algorithms for a lot of graph problems in static scenario, most of the applications in real life deal with networks that are prone to failures. Thus it is essential to come up with graph structures/algorithms that do not lose their functionality on occurrence of failures. This motivates the research on designing fault tolerant structures for various graph problems. In this thesis we provide fault tolerant structures for some of the important graph problems like reachability, strong connectivity, and single source approximate shortest path. Before proceeding to the existing work and our contribution, let us formalize the notion of the fault tolerant model and briefly discuss our goals in this model.

Fault Tolerant Model A failure prone network is modeled as a graph where vertices (or edges) may change their status from active to failed, and vice versa. These failures, though unpredictable, are small in number and are transient due to some simultaneous repair process that is undertaken in these applications. This aspect is captured by associating a parameter k with the network such that there are at most k vertices (or edges) that are failed at any stage, where k is much smaller than the number of vertices in the underlying graph. We assume that the failures are adversarial in

nature, that is, failures occur in an arbitrary order.

Goals in a Fault Tolerant Model Fault tolerance is a broad and rich area, and there can be many aspects to fault tolerance in a network. For example, increasing robustness of a network by adding more links(edges), handling network congestion on occurrence of failures, or designing efficient mechanism of transporting resources to faulty link/node for fast recovery, etc. In this thesis we study fault tolerance from a different perspective. We focus on the design of graph structure that preserve their functionality even after the occurrence of failures. The main objectives addressed in this setting are:

(i) computing a data structure (oracle) that can quickly answer any query for a given problem (e.g. distance, connectivity, etc.) after the occurrence of the failures,

(ii) designing a compact routing scheme in network avoiding the failed nodes/links,

(iii) designing a compact labeling scheme to quickly answer graph queries in the non-centralized setting after the occurrence of failures, and

(iv) computing a sparse subgraph that preserves a certain property (e.g. shortest path, connectedness, etc.) of the graph after the occurrence of the failures.

We now elaborate on these further by taking examples from the static setting.

Oracle. Consider the problem of answering distance queries on a graph in an on-line fashion. The simplest solution of this problem is to compute all-pairs shortest path in $O(mn \log n)$ time and store all distances in $n \times n$ distance matrix. Using this matrix any distance query can be easily answered in $O(1)$ time. However, a major drawback of this solution is that most networks are very large in size, so it is not practical to store the complete $n \times n$ matrix. Thorup and Zwick [TZ05] studied this problem and showed that if we are willing to settle for approximation, then we can get better results for undirected graphs. For instance, they showed that we can compute a graph structure of just $O(n^{3/2})$ size, which can report distances stretched by a factor of at most 3 in constant time. Such a graph data structure is called an oracle and the objective here is to achieve both compactness and fast query time.

When a network is prone to failures, the aim is to compute a data structure that can answer a given query (which here is reporting approximate distance between pair of vertices), not in the input graph G but in the graph $G \setminus F$, where F is the set of failed links/nodes. Such a data structure that is functional even after failures is commonly referred as fault tolerant oracle.

Routing Scheme. In very large networks (like LAN/Internet) there is no centralized processing system, and nodes communicate with each other by passing messages. In such a scenario, each node has its own local processor and in order to facilitate communication it has a small routing table that decides where a message should be forwarded/sent. Thorup and Zwick [TZ01] considered the problem of computing a routing scheme that stores at each node a small routing table and is able to route message from any source node to any destination node over paths stretched by small factor. They showed that for any parameter t, we can construct a routing scheme where nodes have a routing table of $\widetilde{O}(n^{1/t})$ size and route packets over paths that are stretched by a factor of at most $2t-1$. The header attached to the packets in this scheme is of $o(\log^2 n)$-bits.

In a faulty network, the aim is to design routing protocol that, for any given set F of failures, is able to route messages between any pair of nodes in graph $G \setminus F$ efficiently.

Labeling Scheme. For a given graph problem, labeling scheme assigns each vertex u a small label $L(u)$ such that given any query over some vertices, say $v_1, \ldots, v_t$, it can be answered efficiently by only processing the labels $L(v_1), \ldots, L(v_t)$. For answering distance queries in undirected graphs, Alstrup et al. [AGHP16a] designed a labeling scheme with labels of $\big(\frac{\log 3}{2}n + o(n)\big)$-bits that given the labels of any two vertices reports the distance between them in constant time. There also exist labeling schemes for problems like LCA, level-ancestor, and distance queries in a rooted tree with labels of $\widetilde{O}(1)$-bits, see [AHL14, AGHP16b, FGNW16].

In a faulty network, the aim is to design a labeling scheme that, given the labels of the query vertices and the failed nodes/edges, is able to quickly answer the queries associated with a given problem.

Sparse Subgraph. Let $\mathcal{P}$ be any property defined over a graph. A subgraph $H = (V, E_0)$, where $E_0 \subseteq E$, is said to preserve property $\mathcal{P}$ if the property $\mathcal{P}$ is satisfied by subgraph H if and

$\tilde{O}()$ hides the polylogarithmic factors.

only if it is satisfied by graph G. There exist interesting results for various fundamental graph properties, e.g., every k-connected graph has a k-connected subgraph with $O(nk)$ edges [NI92]. When it is not possible to preserve the property exactly, we may have to settle for sparse subgraphs which preserve a given property approximately. For example, there exist sparse subgraphs (spanners) [PS89] that preserve all-pairs shortest paths approximately, there exist subgraphs with $O(n \log n)$ edges that approximates every cut [BK15].

In a failure prone graph our aim is to compute a sparse subgraph H preserving property $\mathcal{P}$ even after occurrence of failures. Precisely, for any set F of failures, $H \setminus F$ should preserve property $\mathcal{P}$ either exactly or approximately with respect to graph $G \setminus F$.

In the last decade, many elegant fault tolerant data structures have been designed for various graph problems such as connectivity [PT07, DP10], shortest paths [BK13, CLPR12, DTCR08], spanners [BCPS15, CLPR10], etc. We discuss these in detail in the next section.

1.2 Existing Work

We now provide a brief summary of some of the existing work in the area of fault tolerant data structures for undirected and directed graphs.

1.2.1 Undirected Graphs

(i) Sparse subgraph preserving all-pairs approximate distances (a.k.a. Spanners)

Given an undirected graph $G = (V, E)$, a subgraph $H = (V, E')$, where $E' \subseteq E$, is said to be an α-spanner of G if for any $u, v \in V$, $dist_H(u, v) \leq \alpha \cdot dist_G(u, v)$. The notion of spanners was first introduced by Peleg and Schäffer [PS89] and Peleg and Ullman [PU89a]. Spanners are known to have applications in routing [PU89b, TZ01], distance oracle [TZ05], distance preservers [BCE05], etc. There is vast literature on construction of spanners, see [EP04, BS07, BKMP10, Che13b]. It is known that for every $t \geq 1$, there exist spanners of stretch of $(2t - 1)$ which have size at most $O(n^{1+1/t})$ [ADD$^+$93]. It is also conjectured that this size-stretch trade-off is tight. In a failure prone network, it is natural to ask if we can compute a fault tolerant spanner, that is, can we compute a subgraph H such that for any subset F of k edges and/or vertices, the subgraph $H \setminus F$

is an α-spanner of $G \setminus F$.

Chechik et al. [CLPR10] were the first to show the existence of sparse fault tolerant spanners for general undirected weighted graphs. For edge failures, they give construction of k-edge fault tolerant $(2t-1)$-spanner of size $O(kn^{1+1/t})$. For vertex failures, they give construction of k-vertex fault tolerant $(2t-1)$-spanners of size $O(k^3t^{k+1}n^{1+1/t}\log^{1-1/t} n)$. Dinitz and Krauthgamer [DK11] showed that for any $t \geq 2$, there exists a simple transformation that can convert a $(2t-1)$-spanner construction with at most $f(n)$ edges into a k-vertex fault tolerant $(2t-1)$-spanner construction with at most $O\big(k^3 \log n\ f(2n/k)\big)$ edges. Applying this transformation to the construction given by [ADD$^+$93] leads to a k-vertex fault tolerant $(2t-1)$-spanner with $O(k^{(2-1/t)}n^{1+1/t}\log n)$ edges, thereby improving the bound of [CLPR10] for vertex failures. For the work on fault tolerant additive spanners, see [BCPS15, BGG$^+$15].

Result	**Stretch**	**Faults**	**Edges/ vertices**	**Size**	**Graph type**
[CLPR10]	$(2t-1)$	k	edges	$O(kn^{1+1/t})$	weighted
[CLPR10]	$(2t-1)$	k	vertices	$O(k^3t^{k+1}n^{1+1/t}\log^{1-1/t} n)$	weighted
[DK11]	$(2t-1)$	k	vertices	$O(k^{(2-1/t)}n^{1+1/t}\log n)$	weighted

Table 1.1: Sparse fault tolerant multiplicative spanners for undirected graphs.

(ii) Oracle for all-pairs approximate distance queries

For all-pairs approximate distances, Baswana and Khanna [BK13] showed that for any positive integer t, an unweighted undirected graph can be processed to compute an oracle that can report $(2t-1)(1+\epsilon)$-approximate distances between any two nodes upon failure of a vertex in $O(t)$ time. The size of their data structure is $O\big(t^5n^{1+1/t}(1/\epsilon^4)\log^3 n\big)$.

For multiple edge failures in weighted graphs, Chechik et al. [CLPR12] showed that if W is the ratio of the heaviest and the lightest weight edge in the graph, then we can compute an oracle of $O\big(k \cdot tn^{1+1/t}\log(n\,W)\big)$ size that after any k failures can report $(8t-2)(k+1)$-stretched distances in $\widetilde{O}(k \log\log W)$ time. Later Chechik et al. [CCFK17] improved this result to obtain $(1+\epsilon)$-approximation at the expense of bigger data structure, for any arbitrary ϵ. The size of their data structure is $O\big(kn^2 \log W(\log n/\epsilon)^k\big)$ and the query time is $\widetilde{O}(k^5 \log\log W)$.

Result	Stretch	Query Time	Faults	Edges/ vertices	Oracle Size	Graph Type
[BK13]	$(2t\text{-}1)(1+\epsilon)$	$O(t)$	one	vertex	$O\big(t^5 n^{1+1/t}$ $(1/\epsilon^4)\log^3 n\big)$	unweighted
[CLPR12]	$(8t\text{-}2)(k+1)$	$\widetilde{O}(k$ $\log\log W)$	k	edges	$O\big(k \cdot tn^{1+1/t}$ $\log(nW)\big)$	weighted
[CCFK17]	$(1+\epsilon)$	$\widetilde{O}(k^5$ $\log\log W)$	k	edges	$O\big(kn^2 \log W$ $(\log n/\epsilon)^k\big)$	weighted

Table 1.2: All-pairs approximate distance oracles for undirected graphs.

(iii) Sparse subgraph preserving distances from single source

Parter and Peleg [PP13] addressed the problem of computing a sparse subgraph that preserves the distances from a designated source s on failure of a single vertex or edge. In particular, they show that for any given unweighted undirected graph G we can compute a subgraph H with $O(n^{3/2})$ edges such that for any vertex v and any failure x, the distance of v from s in the graph $H \setminus \{x\}$ is the same as that in $G \setminus \{x\}$. They also showed that this bound is tight. Parter [Par15] extended this result to dual edge failure by showing an upper bound of $O(n^{5/3})$. She also proved a matching lower bound. While the construction for handling single failure is simple, the construction for two failure is relatively complicated.

Baswana and Khanna [BK13] showed that if one is willing to settle for an approximation then there is a subgraph with $O(n \log n + n/\epsilon^3)$ edges that preserves the distances from s up to a multiplicative stretch of $(1+\epsilon)$ upon failure of a single vertex. Bilò et al. [BGLP14] showed that the result of [BK13] can be sparsified to obtain a subgraph of $O(n/\epsilon^3)$ edges that preserves distances up to a stretch of $(1+\epsilon)$ upon an edge failure. For weighted graphs, Bilò et al. [BGLP14] showed that we can compute a subgraph with $O(n \log n/\epsilon^2)$ edges that preserves $(1+\epsilon)$-shortest path after failure of an edge as well as a vertex, for any arbitrarily small ϵ.

In [BGLP16b], Bilò et al. showed that we can compute sparse fault tolerant subgraph that can handle multiple edge failures. They showed that for any $k \geq 1$, we can compute a subgraph of $O(kn)$ size that after failure of any k edge preserves distance from s up to a multiplicative stretch of $(2k+1)$.

Result	Stretch	Faults	Edges/vertices	Size	Graph type
[PP13]	exact	one	both	$O(n^{3/2})$	unweighted
[Par15]	exact	two	edge	$O(n^{5/3})$	unweighted
[BK13]	$(1+\epsilon)$	one	vertex	$O(n \log n + n/\epsilon^3)$	unweighted
[BK13] & [BGLP14]	$(1+\epsilon)$	one	edge	$O(n/\epsilon^3)$	unweighted
[BGLP14]	$(1+\epsilon)$	one	both	$O(n \log n/\epsilon^2)$	weighted
[BGLP16b]	$(2k+1)$	k	edges	$O(kn)$	weighted

Table 1.3: Sparse subgraphs preserving distances from single source in undirected graphs.

Result	Stretch	Faults	Edges/ vertices	Size	Graph type
[PP13]	exact	one	both	$\Omega(n^{3/2})$	unweighted
[Par15]	exact	k	edges	$\Omega(n^{2-1/(k+1)})$	unweighted

Table 1.4: Lower bounds for sparse subgraphs preserving distances from single source.

(iv) Single source distance oracle

Baswana and Khanna [BK13] showed that an unweighted undirected graph can be preprocessed to build an oracle of $O(n \log n + n/\epsilon^3)$ size which for any two vertices v, x, reports in $O(1)$ time a $(1+\epsilon)$-approximate distance from s to v in the graph $G \setminus \{x\}$. For weighted graphs, they showed construction of an $O(n \log n)$ size oracle computable in $O(m \log n + n \log^2 n)$ time which can report 3-approximate distance from the source s in $O(1)$ time. For edge failures, Bilò et al. [BGLP16a] showed that we can construct an oracle of $O(n)$ size which is able to report 2-approximate distances from the source in $O(1)$ time. They also showed that for any $0 < \epsilon < 1$, we can compute an oracle having size $O\big(\epsilon^{-1} n \log(1/\epsilon)\big)$ that can report $(1+\epsilon)$-stretched distances in $O\big(\epsilon^{-1} \log n \log(1/\epsilon)\big)$ time.

All the results stated above are for single edge or single vertex failure only. For multiple edge failures, Bilò et al. [BGLP16b] showed that that we can compute a data structure of $O(kn \log^2 n)$ size that after any k edge failures is able to report the $(2k+1)$-stretched distance from s in $O(k^2 \log^2 n)$ time.

Result	Stretch	Query Time	Faults	Edges/ vertices	Oracle Size	Graph Type
[BK13]	$(1+\epsilon)$	$O(1)$	one	vertex	$O\big(n\log n+ n/\epsilon^3\big)$	unweighted
[BK13]	3	$O(1)$	one	vertex	$O(n\log n)$	weighted
[BGLP16a]	2	$O(1)$	one	edge	$O(n)$	weighted
[BGLP16a]	$(1+\epsilon)$	$O\Big(\frac{\log n}{\epsilon}\log\frac{1}{\epsilon}\Big)$	one	edge	$O\Big(\frac{n}{\epsilon}\log\frac{1}{\epsilon}\Big)$	weighted
[BGLP16b]	$(2k+1)$	$O(k^2\log^2 n)$	k	edges	$O(kn\log^2 n)$	weighted

Table 1.5: Single source distance oracles for undirected graphs.

(v) Connectivity oracle

In the last decade several researchers studied fault tolerant connectivity in undirected graphs. Pătraşcu and Thorup [PT07] presented a data structure of $O(m)$ size that can handle any k edge failures in $O(k\log^2 n\log\log n)$ time to subsequently answer connectivity queries between any two vertices in $O(\log\log n)$ time. For small values of k, Duan and Pettie [DP10] improved the update time of [PT07] to $O(k^2\log\log n)$ by presenting a data structure of $\widetilde{O}(m)$ size.

For handling vertex failures, Duan and Pettie [DP17] showed that there exists a data structure of $O(mk\log n)$ size with $O(k^3\log^3 n)$ update time and $O(k)$ query time. They also presented a Monte-Carlo algorithm which has a better update time of $O(k^2\log^5 n)$ and can answer connectivity queries correctly with high probability in $O(k)$ time. The size of their data structure is $O(m\log^4 n)$.

Result	Time for handling k failures	Query Time	Faults	Edges/ vertices	Oracle Size
[PT07]	$O(k\log^2 n\log\log n)$	$O(\log\log n)$	k	edges	$O(m)$
[DP10]	$O(k^2\log\log n)$	$O(\log\log n)$	k	edges	$\widetilde{O}(m)$
[DP17]	$O(k^3\log^3 n)$	$O(k)$	k	vertices	$O(mk\log n)$
[DP17]	$O(k^2\log^5 n)$	$O(k)$	k	vertices	$O(m\log^4 n)$

Table 1.6: Connectivity oracles for undirected graphs.

(vi) Fault tolerant all-pairs routing

Recall that for the static case, the best currently known routing scheme for undirected weighted graphs is due to Thorup and Zwick [TZ01], that stores at each node a routing table of $\widetilde{O}(n^{1/t})$ size and routes packet over paths that are stretched by a factor of at most $2t-1$. Chechik et al. [CLPR12] gave a construction of fault tolerant routing scheme for undirected weighted graphs that for any two vertices $u, v \in V$ and any set F of at most two failed edges, routes the packet from u to v over a path whose length is at most $O(t)$ times the length of the shortest path in $G \setminus F$. The sum total size of all the routing tables in this scheme is $O(tn^{1+1/t}\log(nW)\log n)$. Chechik [Che13a] later generalized this result to arbitrary k-edge failures. She gave the construction of a routing scheme that after k failures, could route packets over $O\big(t \cdot k^2(k+\log^2 n)\big)$-stretched paths. Each node v in this scheme has a label of $O(\log(nW)\log n)$-bits and a routing table of size at most $O\big(tn^{1/t}deg(v)\log(nW)\log^2(n)\big)$-bits. The scheme uses $O(k\log n)$-bits header that is attached to all the packets. For the case of single edge failure, Rajan [Raj12] showed that we can achieve small stretch as well as small routing table per node. He gave construction of a routing scheme that had a stretch of $O(t^2)$, header of $O(t\log n+\log^2 n)$-bits, and a routing table $\widetilde{O}\big(n^{1/t}+t\cdot deg(v)\big)$-bits for any node v.

Result	**Stretch**	**Faults**	**Edges/ vertices**	**Size**
[CLPR12]	$O(t)$	2	edges	$\widetilde{O}(tn^{1+1/t}\log(nW))$ overall
[Che13a]	$O\big(t\cdot k^2(k+\log^2 n)\big)$	k	edges	$\widetilde{O}\big(tn^{1/t}deg(v)\log(nW)\big)$-bits per vertex v
[Raj12]	$O(t^2)$	1	edge	$\widetilde{O}\big(n^{1/t}+t\cdot deg(v)\big)$-bits per vertex v

Table 1.7: All-pairs fault tolerant routing for weighted undirected graphs.

1.2.2 Directed Graphs

(i) All-pairs distance oracle

For the problem of reporting distances between any arbitrary pair of vertices in directed graphs, Demetrescu et al. [DTCR08] gave a construction of $O(n^2\log n)$ size data structure that for any

$u, v \in V$ and any failed edge/vertex x reports the length of the shortest path from u to v avoiding x in constant time. Duan and Pettie [DP09] extended the result of [DTCR08] to dual failures by designing a data structure of $O(n^2 \log^3 n)$ space that can answer any distance query upon two edge/vertex failures in $O(\log n)$ time. The authors of [DP09] comment that their techniques do not seem to be extensible beyond two failures, and even an oracle for three failures seem too hard to achieve.

Result	**Stretch**	**Query Time**	**Faults**	**Edges/ vertices**	**Oracle Size**	**Graph Type**
[DTCR08]	exact	$O(1)$	one	both	$O(n^2 \log n)$	weighted
[DP09]	exact	$O(\log n)$	two	both	$O(n^2 \log^3 n)$	weighted

Table 1.8: All-pairs distance oracles for directed graphs.

(ii) Sparse subgraph preserving distances from single source

Parter and Peleg [PP13] showed that for any unweighted directed graph G with a designated source vertex s, we can compute a subgraph with $O(n^{3/2})$ edges that preserves the distances from s after failure of any single edge or vertex.

Result	**Faults**	**Edges/vertices**	**Size**	**Graph type**
[PP13]	one	both	$O(n^{3/2})$	unweighted

Table 1.9: Single source distance preserving subgraphs for directed graphs.

(iii) Single source reachability oracle and subgraph

The work on single fault tolerant reachability follows as a by-product of the seminal work of [LT79] on dominators. Given any directed graph G we can compute a reachability preserving subgraph H such that subsequent to failure of any edge/vertex x the set of vertices reachable from source s in graph $H \setminus \{x\}$ is the same as in the graph $G \setminus \{x\}$. Moreover, from the concept of dominator tree presented in [LT79] we can also obtain an $O(n)$ size oracle that for any x, v can answer in $O(1)$ time if v is reachable from s in $G \setminus \{x\}$.

Result	Problem	Faults	Edges/vertices	Size	Graph type
[LT79]	sparse subgraph	one	both	$2n$	any
[LT79]	reachability oracle	one	both	$O(n)$	any

Table 1.10: Single source reachability oracles and subgraphs for directed graphs.

(iv) Oracle for reporting strongly connected components (SCCs)

Georgiadis, Italiano and Parotsidis [GIP17] considered the problem of reporting SCCs of a graph after failure of a single edge or vertex. They showed that it is possible to preprocess G in $O(m+n)$ time to obtain an $O(n)$ size data structure that for any failed edge/vertex x computes the SCCs of the graph $G \setminus \{x\}$ in $O(n)$ time. They also presented a construction of an $O(n)$ size oracle that can answer in $O(1)$ time whether any two given vertices of the graph are strongly connected or not after the failure of any given single edge or vertex.

1.2.3 Fault Tolerance: Undirected versus Directed Graphs

From the existing work it can be seen that very little research has been done in the area of fault tolerant data structures for directed graphs. One of the major reasons behind this is that finding small size subgraphs that preserve distance related properties or computing a compact and efficient distance oracle is a hard task in directed graphs. This is because many of the standard tools that are being used in undirected graphs rely on the fact that distances in undirected graphs are symmetric which is obviously not the case in directed graphs.

In fact, in some cases efficient data structures for directed graphs are not even possible. Graph spanners are probably the most notable example for a separation between undirected and directed graphs. For every weighted undirected graph there is a subgraph with $O(n^{1+1/k})$ edges that approximates all-pair distances with a multiplicative stretch of $2k-1$ [PS89]. It is very easy to see that such a general result is not possible for directed graphs since no subgraph can approximate distances between all pairs in a complete bipartite graph.

In directed graphs, there are some problems, like reachability preserving subgraph, strong connectivity oracle, etc. for which fault tolerant data structures exist but only for single failure

[LT79, GIP17]. So it would be interesting to see if these results can be extended to multiple failures.

There are also problems where efficient data structures exists for undirected graphs but it is not known if the same is possible for directed graphs. For example, for the problem of approximate shortest path from a designated source there exist results for handling both single as well as multiple failures in undirected graphs [BK13, BGLP16a, BGLP16b], but until now no such work has been carried out for directed graphs.

In a distributed environment, it is important to design a routing scheme that is resilient to failures in directed graphs. Specifically, the Autonomous System's link-state database, that is used by the Open Shortest Path First (OSPF) TCP/IP internet routing protocol, is a directed graph [Moy99].

1.3 Our Contributions

This thesis contributes the following new fault tolerant data structures for directed graphs.

- An alternative $O(m \log n)$ time algorithm for computing dominators.
- A compact single source 2-fault tolerant reachability oracle and labeling scheme.
- A sparse subgraph preserving reachability from a single source after k (≥ 1) failures.
- An oracle for reporting SCCs of a directed graph after k (≥ 1) failures.
- A sparse subgraph, an oracle, a labeling scheme, and a routing scheme for $(1+\epsilon)$-approximate distances from a single source to handle single failure.

We now elaborate these results. In the first result, we present a simple and alternative algorithm for computing dominators that takes $O(m \log n)$ time. Given a directed graph G and a designated source s, a vertex x is said to be a dominator of a vertex v if every path from s to v contains x. The single source 1-fault tolerant reachability is closely related to the notion of dominators. Though ours is not an optimal algorithm and there exist better results like [LT79, BGK$^+$08], but it is the first result that does not exploit a DFS tree for computing dominators.

In the next result, we present an optimal solution to the problem of reporting reachability information from a designated source s under dual vertex failures. We show that it is possible

to compute in polynomial time an $O(n)$ size data structure that for any query vertex v, and any pair of failed vertices f_1, f_2, answers in $O(1)$ time whether or not there exists a path from s to v in $G \setminus \{f_1, f_2\}$. We also present a distributed implementation of this oracle (that is, a labeling scheme).

In our result on sparse reachability preserving subgraph, we show that for every $k \geq 1$ there is a sparse subgraph H of G with at most $2^k n$ edges that preserves the reachability from s after any k failures. We call H a k-Fault Tolerant Reachability Subgraph (k-FTRS). Formally, H is said to be a k-FTRS if for any set F of k edges (vertices), a vertex $v \in V$ is reachable from s in $G \setminus F$ if and only if v is reachable from s in $H \setminus F$. Until recently, a sparse FTRS was known to exist only for single failure, and this result follows from the work of Lengauer and Tarjan [LT79] on dominators. We also show that the upper bound of $2^k n$ is tight by providing a matching lower bound.

Using our k-FTRS structure, we obtain an oracle for strong connectedness under multiple failures. We show that for any directed graph G on n vertices, and any integer $k \geq 1$, there is a data structure that computes in $O(2^k n \log^2 n)$ time all the strongly connected components of the graph $G \setminus F$, for any set F of size at most k. The time taken to report the strongly connected components is almost optimal since the time for outputting the SCCs of $G \setminus F$ is at least $\Omega(n)$.

For the problem of fault tolerant approximate shortest path, we present various results, namely, a subgraph, an oracle, a labeling scheme, and a routing scheme for maintaining approximate single source shortest paths from a designated source s under single failure. We show that for any directed weighted graph G with edge weights in the range $[1, W]$, we can compute a subgraph H with $\widetilde{O}(n \log_{1+\epsilon}(n\,W))$ edges such that for any $x, v \in V$, the graph $H \setminus x$ contains a path whose length is a $(1+\epsilon)$-approximation of the shortest path from s to v in $G \setminus x$. Using this subgraph, we present a single source routing scheme that can route on a $(1+\epsilon)$-approximation of the shortest path from a fixed source s to any destination t in the presence of a fault. We also obtain an efficient oracle and a compact labeling scheme for reporting $(1+\epsilon)$-approximate distances after the occurrence of the failure.

1.4 Organization of the Thesis

In Chapter 2, we give the alternate algorithm for computing domintaors. In Chapter 3, we present the dual fault tolerant reachability oracle and also present a labeling scheme for this problem. In chapter 4, we give the construction of k-fault tolerant reachability subgraph and also present a matching lower bound. In Chapter 5, we use our k-FTRS structure to obtain an oracle for reporting strongly connected components of a graph after any k failures. We turn to the problem of preserving $(1+\epsilon)$-approximate distances in Chapter 6, and present our construction for a sparse subgraph, an oracle, a labeling scheme, and a compact routing scheme.

Chapter 2

An Alternate Algorithm for Computing Dominators

Dominators are known to have a lot of applications including fault-tolerance. Thus it is important to design efficient algorithms for computing them.

The single fault tolerant reachability is closely related to the notion of *dominators* as follows. Given a directed graph G and a source vertex s, we say that a vertex x dominates a vertex v if every path from s to v contains x. Lengauer and Tarjan [LT79] introduced a data structure called *dominator tree* which is a tree rooted at s such that for any v in G, the set of ancestors of v in the tree is precisely the set of dominators of v. Thus, for any two vertices x and v in G, v becomes unreachable from s on failure of x if and only if x is ancestor of v in the dominator tree.

A lot of work has been done on computing dominators in optimal and near-optimal time, see [LT79, BGK$^+$08, GT12]. However, one thing that is common in all these results is that they are based on DFS tree and crucially use its properties to efficiently compute dominators. In fact, the only non-trivial result that could compute dominators without a DFS tree earlier was for directed acyclic graphs (DAGs) [FGMT13]. Thus, it natural to ask - “Can we efficiently compute dominators in general directed graphs without employing a DFS structure?" In this chapter, we affirmatively answer this question by providing a near optimal construction for dominators that works for general graphs.

2.1 Definitions and Notations

Let T be any arbitrary tree of G rooted at s and let $\mathcal{P}$ denote a sequence of vertices in G defined by any pre-order traversal of tree T. For notational convenience, throughout this chapter, a vertex will also be denoted by its pre-order numbering. Thus, $u < v$ would imply that the pre-order number of u is less than that of v.

Definition 2.1 (Dominator)**.** *A vertex w is said to be a dominator of a vertex v in a directed graph G with s as designated source if every path from s to v passes through vertex w.*

Definition 2.2 (Immediate Dominator)**.** *A vertex w is said to be the immediate dominator of v, denoted by $w = \text{IDOM}(v)$, if w is a dominator of v and every dominator of v (other than v itself) is also a dominator of w.*

Definition 2.3. *A simple path $P = (u_0, \ldots, u_t = v)$ in G is said to be a detour of v with respect to T if u_0 is an ancestor of v, and for $0 < i < t$, none of the u_i's is an ancestor of v in T.*

Definition 2.4. *A detour to v with respect to T that emanates from the ancestor of v with minimal pre-order number is called a* highest detour *of v.*

We denote the highest detour of v with $\text{HD}(v)$. In case that there is more than one highest detour we pick one arbitrarily.

The reachability tree T induces a classification of edges of G into four categories. These are as follows:

1. *Tree Edges*: Any edge in G that appears in the tree T is called a tree edge.

2. *Forward Edges*: An edge (u, v) where u is an ancestor of v in T, but not the parent of v is called a forward edge.

3. *Back Edges*: An edge (u, v) where u is a descendant of v in T is called a back edge.

4. *Cross Edges*: All the remaining edges are categorized as cross edges. The endpoints of these edges do not have any ancestor-descendant relationship in T.

2.2 DFS Tree versus Arbitrary Tree

Lengauer and Tarjan [LT79] presented an algorithm for computing immediate dominators. As part of their work they defined semidominators over a DFS tree T. Their definition of semidominators can be reformulated as follows:

Definition 2.5 (Semidominators in a DFS Tree)**.** *Given a DFS tree, the semidominator of v ($v \neq s$) is defined to be the highest ancestor of v from which there is a detour to v. We use the notation* $\text{SDOM}(v)$ *to denote the semidominator of v.*

For the case when T is a DFS tree, Lengauer and Tarjan [LT79] gave an $O(m\alpha(m,n))$ time algorithm for computing immediate dominators. In order to prove this bound they used a crucial property of DFS tree which in simple words can be re-stated as follows.

Property 2.1. *If $(\text{SDOM}(v), v)$ is not an edge in G then we can always find a highest detour* $\text{HD}(v)$ *for v which can be represented as $\langle \text{HD}(w) :: \text{PATH}_T(w,y) :: (y,v)\rangle$, where y is an in-neighbor of v and w is either equal to y or an ancestor of y.*

For the case when T is an arbitrary tree, and not a DFS tree, Property 2.1 no longer holds. A simple example that illustrates the situation for general trees is shown in Figure 2.1. Thus it is not immediately clear whether we can compute immediate dominators by starting with an arbitrary tree. In order to achieve our goal, we define semidominators for arbitrary trees in the next section.

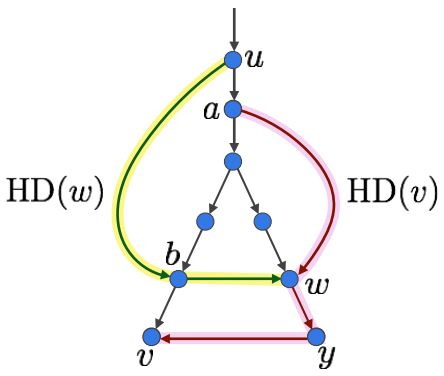


Figure 2.1: The paths highlighted in yellow and pink colors respectively represent the highest detours $\text{HD}(w)$ and $\text{HD}(v)$ for vertices w and v. The detour $\text{HD}(v)$ cannot be expressed as $\langle \text{HD}(w) :: \text{PATH}_T(w,y) :: (y,v)\rangle$ because $\text{HD}(w)$ passes through an ancestor of v.

2.3 Semidominators with respect to Arbitrary Tree

Given an arbitrary tree T, let D be a detour from a vertex u to a vertex v with *minimum* number of non-tree edges. Let (u_1, v_1) be the first edge in D and let $(u_2, v_2), (u_3, v_3), \ldots, (u_t, v_t)$ be the sequence of non-tree edges in the order they appear in $D \backslash (u_1, v_1)$. Here $u_1 = u$ and $v_t = v$. Consider the edge (u_i, v_i), where $1 < i \leq t$. Since the segment of D from v_{i-1} to u_i is a path in T it follows that $u_i \in T(v_{i-1})$. Moreover, $v_i \notin T(v_{i-1})$ because if $v_i \in T(v_{i-1})$ we can replace the segment of D from v_{i-1} to v_i by $\text{PATH}_T(v_{i-1}, v_i)$, thereby reducing the number of non-tree edges.

Consider now the sequence - $(u, v_1, v_2, \ldots, v_t)$. From the above discussion it follows that vertices in this sequence satisfy the relation that $v_1 \in \text{OUT}(u)$, and for $1 < i \leq t$, $v_i \in \text{OUT}(T(v_{i-1}))$. This motivates us to define the notion of a valid sequence as follows.

Definition 2.6 (Valid sequence)**.** *A sequence of vertices $(u, v_1, v_2, \ldots, v_t = v)$ is said to be a valid sequence with respect to tree T if the following two conditions hold:*

(i) (u, v_1) *is an edge in G.*

(ii) for $1 < i \leq t$, $v_i \in \text{OUT}(T(v_{i-1}))$.

Let u, v be any two vertices such that u is an ancestor of v in T. It follows from Definition 2.6 that if there exists a detour from u to v in T, then there exists a valid sequence from u to v. However, the other direction is not always true, that is, a valid sequence from u to v in T may not correspond to a detour in T. For example, consider the sequence $\sigma = (u, b, w, v)$ in Figure 2.1. This is a valid sequence but there is no detour from u to v. The one to one correspondence between detours and valid sequences holds only when T is a DFS tree.

We will now define semidominator with respect to an arbitrary tree using valid sequence. In arbitrary trees, it will turn out that if there is a valid sequence from $\text{SDOM}(v)$ to v and $\text{SDOM}(v) \neq parent_T(v)$, then there are two vertex disjoint paths from $\text{SDOM}(v)$ to v.

Definition 2.7 (Semidominators in an Arbitrary Tree)**.** *A vertex u is semidominator of v if (i) u is an ancestor of v, (ii) there is a valid sequence from u to v, and (iii) there is no other vertex on* $\text{PATH}_T(s, u)$ *which has a valid sequence to v.*

The following lemma shows that in the case of a DFS tree, Definition 2.7 degenerates to Definition 2.5.

Lemma 2.1. *Let T' be a DFS tree of G, and $u, v \in T'$ be such that u is an ancestor of v and there exists a valid sequence from u to v in G. Then there must also exist a detour from u to v.*

Proof. Let $\sigma = (u, v_1, v_2, \ldots, v_t = v)$ be a valid sequence from u to v. For $i \in [2, t]$, let $e_i = (u_i, v_i)$ be an edge emanating out from the subtree $T'(v_{i-1})$ and terminating to v_i. Now consider the path

$$D = (u, v_1) :: \Big(\textsc{Path}_{T'}(v_1, u_2) :: (u_2, v_2)\Big) :: \ \cdots \ :: \Big(\textsc{Path}_{T'}(v_{t-1}, u_t) :: (u_t, v_t)\Big)$$

We will show that D is a detour from u to v, that is, none of the internal vertices of D lie on $\textsc{Path}_{T'}(s, v)$. Since for $i \in [2, t]$, the segment of D from v_{i-1} to u_i is a tree path, it suffices to show that the vertices - $v_1, v_2, \ldots, v_{t-1}$ are not an ancestor of v.

Consider the edge (u_i, v_i) emanating out from the subtree $T'(v_{i-1})$, for some $i \in [2, t]$. As (u_i, v_i) is either a cross edge or a back edge, it follows from the properties of DFS traversal that $\textsc{Visit-Time}(v_i) < \textsc{Visit-Time}(v_{i-1}) \leq \textsc{Visit-Time}(u_i)$. On applying this inequality recursively, we get that for each $i < t$, $\textsc{Visit-Time}(v_t) < \textsc{Visit-Time}(v_i)$. This shows that none of the internal vertices of σ can be an ancestor of v, and thus D is a detour from u to v in G. □

We now show some of the properties of semidominators defined with respect to any arbitrary tree.

Lemma 2.2. *Let $u, v, w \in V$ be three vertices such that $v \in \textsc{out}(T(w))$, $v \notin \textsc{out}(u)$, and u is some common ancestor of v and w. If G contains two vertex disjoint paths from u to w, then it also contains two vertex disjoint paths from u to v.*

Proof. Let us assume towards a contradiction that G does not contain two vertex disjoint paths from u to v. Then it follows from Menger's Theorem [Wil86] that there exists a vertex x (other than u, v) such that every path from u to v in G passes through x. So vertex x lies on $\textsc{Path}_T(u, v)$. Let P and Q be two vertex disjoint paths from u to w in G (see Figure 2.2). Since $w \neq x$, at least one out of these two paths, say Q, does not pass through x. Now $\langle Q :: \textsc{Path}_T(w, y) :: (y, v)\rangle$ gives a path from u to v not passing though x. (Though this concatenated path may contain loops,

but we can remove all these loops). Thus we get a contradiction. So G must contain two vertex disjoint paths from u to v. □

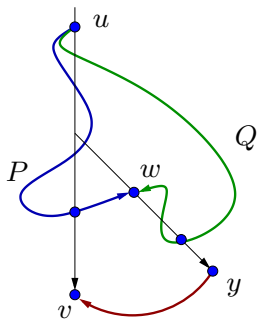


Figure 2.2: Two vertex disjoint paths P and Q from u to w

Lemma 2.3. *Let $u, v \in V$ be such that there is no ancestor-descendant relationship between u and v in tree T. If there exists a valid sequence from u to v in G, then there exists a valid sequence from $\text{LCA}(u, v)$ to v as well.*

Proof. Let $\sigma = (u, v_1, v_2, \ldots, v_t = v)$ be a valid sequence from u to v, and u_0 be $\text{LCA}(u, v)$. Let y be the child of u_0 lying on $\text{PATH}_T(u_0, u)$. Let v_j be the first vertex of the sequence σ that doest not lie in $T(y)$. If $j = 1$, then the in-neighbor u of v_1 lies in $T(y)$. If $j > 1$, then v_j has an in-neighbor, say u_j, that lies in $T(v_{j-1}) \subseteq T(y)$. Thus, $v_j \in \text{OUT}(T(y))$. Therefore, the sequence $\sigma = (u_0, y, v_j, v_{j+1}, \ldots, v_t)$ is a valid sequence from $\text{LCA}(u, v)$ to v. □

The following corollary provides an alternative definition to semidominators. We shall use these two definitions interchangeably henceforth.

Corollary 2.1. *Let u be a vertex with minimum pre-order number such that there exists a valid sequence from u to v. Then u is an ancestor of v, and therefore a semidominator of v.*

The following theorem shows that whenever $\text{SDOM}(v) \neq parent_T(v)$, then G contains two vertex disjoint paths from $\text{SDOM}(v)$ to v.

Theorem 2.1. *Let $v \in V$ be such that $\text{SDOM}(v) \neq parent_T(v)$. Then there exist two vertex disjoint paths from $\text{SDOM}(v)$ to v in G.*

Proof. Let $\sigma = (u, v_1, v_2, \ldots, v_t = v)$ be a valid sequence from $u = \text{SDOM}(v)$ to v. We fist show that all the vertices of σ lie in the subtree $T(u)$. Let us assume on the contrary that there exists a

vertex v_j ($1 \leq j \leq t$) that lies outside the subtree $T(u)$. Notice that $(parent_T(v_j), v_j, v_{j+1}, \ldots, v_t)$ is also a valid sequence to v. So on applying Lemma 2.3 we get that there exists a valid sequence from $\text{LCA}(parent_T(v_j), v)$ to v. Since $\text{LCA}(parent_T(v_j), v)$ is an ancestor of $u = \text{SDOM}(v)$, this violates the fact that u is a semidominator of v. Hence we get a contradiction and thus all the vertices of σ must lie in the subtree $T(u)$.

As all the vertices of σ lies in $T(u)$, from Lemma 2.2 it follows that if there exists two vertex disjoint paths from s to some vertex v_j ($j < t$), then there will exist two vertex disjoint paths from s to v_{j+1} as well. Thus it suffices to show that there exist two vertex disjoint paths from u to v_1 or from u to v_2. If (u, v_1) is a forward edge, then it is easy to see that the edge (u, v_1) and the tree path $\text{PATH}_T(u, v_1)$ are two vertex disjoint paths from u to v_1. Let us now consider the case when (u, v_1) is a tree edge. Let y be an in-neighbor of v_2 lying in the subtree $T(v_1)$. Then in this case $\text{PATH}_T(u, v_2)$ and (u, v_1)::$\text{PATH}_T(v_1, y)$::(y, v_2) are two vertex disjoint paths from u to v_2. □

2.4 Algorithm for Computing Semidominators and Valid sequences

In this section, we provide an efficient algorithm for computing the semidominator and a corresponding valid sequence for each $v \in V \setminus \{s\}$ with respect to a given tree T. Throughout this section, we use the notation $\sigma(v)$ to denote a valid sequence from $\text{SDOM}(v)$ to v. Our algorithm iteratively processes the vertices of G in the increasing order of the pre-order numbering $\mathcal{P}$. Let v_i denote the vertex at the i^{th} place in $\mathcal{P}$. During i^{th} iteration, the algorithm computes the set W_i consisting of all those vertices whose semidominator is v_i.

Consider a vertex v_i. Let B denote the set of all those vertices w for which there exists a valid sequence starting from v_i and ending at w. The set B can be computed as follows. We initialize B as the out-neighbors of v_i. Next we add $\text{OUT}(T(w))$ to B for each w in B, and proceed recursively. By the alternative definition of semidominators as in Corollary 2.1 we have that $W_i = B \setminus (\cup_{j<i} W_j)$.

In order to design an efficient implementation of the algorithm outlined above, there are two requirements. The first requirement is that while computing valid sequences from v_i we should not process those vertices whose semidominator have already been computed. For this purpose, we keep a flag variable $active/inactive$ corresponding to each vertex w in G. At any instant of

time the active vertices are those vertices whose semidominator has not yet been computed. The second requirement is that given any vertex w we should be able to compute the set of active nodes in $\text{OUT}(T(w))$ efficiently. In order to fulfill these requirements, we use a data structure $\mathcal{D}$ that supports the following two operations efficiently.

1. ACTIVEOUTNGHBRS($\mathcal{D}, T(w)$): return the set of active nodes in $\text{OUT}(T(w))$.

2. MARKINACTIVE($\mathcal{D}, S$): mark the vertices in set S as inactive. This is done by simply deleting from $\mathcal{D}$ the incoming edges to all vertices present in set S.

The data structure $\mathcal{D}$ is a suitably augmented segment tree formed on an Euler tour of the tree T. The data structure takes $O(deg(A) \log n))$ time to perform ACTIVEOUTNGHBRS($\mathcal{D}, T(w)$) operation, where A is the set of vertices reported. It takes $O(deg(S) \log n)$ time to perform MARKINACTIVE($\mathcal{D}, S$) operation. We provide the complete details of the data structure in Subsection 2.4.1.

Algorithm 2.1: Computing semidominator and the corresponding valid sequence

```
1  Q ← ∅;
2  for i = 1 to n do
3  |  S ← Set of active vertices lying in OUT(v_i);
4  |  ENQUEUE(Q, S);
5  |  MARKINACTIVE(D, S);
6  |  foreach w ∈ S do σ(w) = (v_i, w);
7  |  while (Q ≠ ∅) do
8  |  |  x ← DEQUEUE(Q);
9  |  |  SDOM(x) ← v_i;
10 |  |  S ← ACTIVEOUTNGHBRS(D, T(x));
11 |  |  ENQUEUE(Q, S);
12 |  |  MARKINACTIVE(D, S);
13 |  |  foreach w ∈ S do σ(w) = (σ(x), w);
14 |  end
15 end
```

Algorithm 2.1 gives the pseudo code for computing semidominators. It maintains a queue $\mathcal{Q}$ throughout the run of algorithm. The semidominator of the vertices is computed in the order they are enqueued. Initially all the vertices in G except root are marked as active. A vertex is marked inactive as soon as it is enqueued in $\mathcal{Q}$. In the i^{th} iteration the algorithm computes the set of all those vertices whose semidominator is v_i as follows. First it computes the set S of all the active

out-neighbors of v_i. This set is enqueued and for each $w \in S$, $\sigma(w)$ is set as (v_i, w). Next while $\mathcal{Q}$ is non empty, it removes the first vertex say x from $\mathcal{Q}$. For each active node w in $\text{OUT}(T(x))$, $\sigma(w)$ is assigned as $(\sigma(x), w)$ and w is enqueued in $\mathcal{Q}$. This process is repeated until $\mathcal{Q}$ becomes empty. Vertex v_i is assigned as semidominator of all the vertices enqueued in the i^{th} iteration.

Figure 2.3 illustrates the execution of our algorithm. Figure 2.3(a) depicts the first iteration which is supposed to compute W_1. The vertices that are enqueued before the while loop are $\langle 2, 14 \rangle$. The execution of the while loop will place vertices 15 and 16 into the queue in this order. It can be visually inspected that these vertices constitute W_1. Similarly Figure 2.3(b) depicts the second iteration that is supposed to compute W_2. The vertices that are enqueued before entering the while loop are $\langle 3, 7, 12 \rangle$. The execution of the while loop will place vertices 5, 10, 4, 8, 11 into the queue in this order. It can be visually inspected that these vertices constitute W_2.

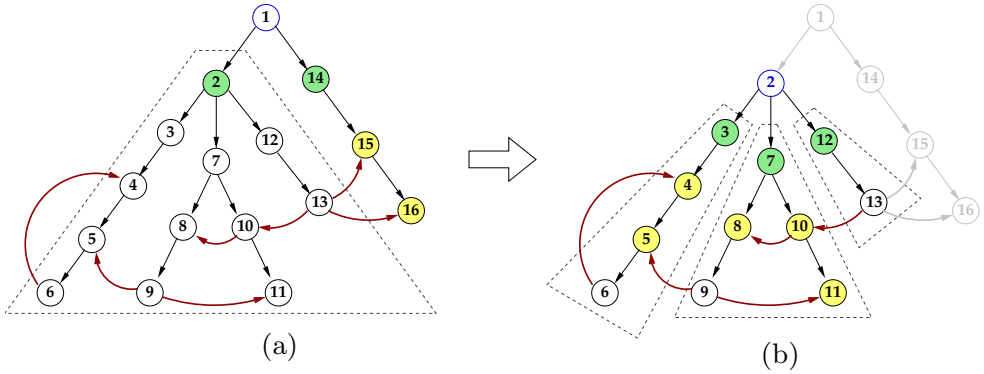


Figure 2.3: The filled vertices in Figure (a) and (b) respectively constitute the sets W_1 and W_2. Figure (b) shows that all the vertices in W_1 are marked inactive in round 2.

We now analyze the time complexity of Algorithm 2.1. The total time taken by Step 3 in the algorithm is $O(m)$. The time taken by steps 5, 10, and 12 is $O(\log n)$ times the sum of degrees of vertices enqueued in $\mathcal{Q}$. Since each vertex is enqueued at most once, the running time of the algorithm is $O(m \log n)$.

Theorem 2.2. *There exists an $O(m \log n)$ time algorithm that for any given graph with n vertices and m edges, and any given reachability tree T rooted at source, computes the semidominator and a minimum length valid sequence for each vertex in G.*

2.4.1 Data Structure

Let $\mathcal{T}$ be a segment tree [Ben77] whose leaf nodes from left to right correspond to the sequence $\langle v_1, ..., v_n \rangle$ (see Figure 2.4). Our data structure will be $\mathcal{T}$ whose nodes are suitably augmented as follows. Let (u, v) be an edge in G. We store a copy of the edge as the ordered pair (u, v) at all ancestors of u (including itself) in tree $\mathcal{T}$. Thus each edge in G is stored at $O(\log n)$ levels in $\mathcal{T}$. Let $\mathcal{E}(b)$ be the collection of edges stored at any node b in $\mathcal{T}$. We keep the set $\mathcal{E}(b)$ sorted by the second endpoint of the edges in a doubly link list. For each edge $(u, v) \in E$, we also store pointer to all $\log n$ copies of it in $\mathcal{T}$. The size of the data structure is $O(m \log n)$ in the beginning.

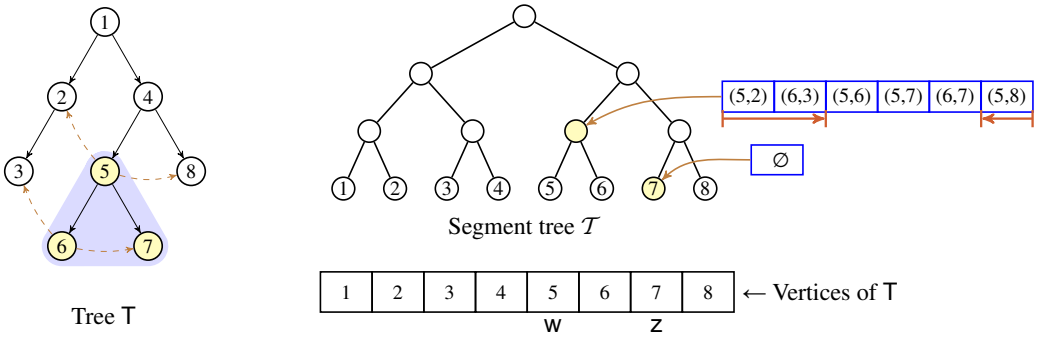


Figure 2.4: Data structure

The operation MARKINACTIVE($\mathcal{D}, S$) involves deletion of incoming edges to all the vertices in set S. Since we store pointers to all $\log n$ copies of an edge, a single edge can be deleted from the data structure in $O(\log n)$ time. So the time taken by this operation is $O(deg(S) \log n)$.

We now show that $\mathcal{D}$ can perform the operation ACTIVEOUTNGHBRS($\mathcal{D}, T(w)$) quite efficiently. Let S_0 be the set of active nodes in OUT($T(w)$). Note that the pre-order numbering of the vertices in $T(w)$ will be a contiguous subsequence of $[1, .., n]$, and w would be the vertex of minimum pre-order number in $T(w)$. Let z be the vertex with maximum pre-order number in subtree $T(w)$ (This information can be precomputed in total $O(n)$ time for all vertices in the beginning). So $[w, .., z]$ denotes the set of vertices in $T(w)$.

Notice that any contiguous subsequence of $[1, .., n]$ can be expressed as disjoint union of at most $\log n$ subtrees in $\mathcal{T}$. Let $\tau_1, ..., \tau_\ell$ denote these subtrees for the subsequence $[w, .., z]$. For $i = 1$ to ℓ, let E_i denote the set of all those edges (x, y) such that x is a leaf node of τ_i and y lies outside the set $[w, .., z]$. It can be observed that the desired set S_0 corresponds to the set of second-endpoints of all edges in the set $\cup_{i=1}^{\ell} E_i$. Let $b_1, ..., b_\ell$ respectively denote the roots of subtrees

$\tau_1, ..., \tau_\ell$ in $\mathcal{T}$. Then set E_i can be computed by scanning the list $\mathcal{E}(b_i)$ from beginning (and respectively end) till we encounter an edge (u, v) with v lying in range $[w, .., z]$. (See Figure 2.4). Thus the time taken by the operation ACTIVEOUTNGHBRS($\mathcal{D}, T(w)$) is bounded by $O(deg(S_0) + \log n)$, where S_0 is the set of vertices reported.

This data structure can be preprocessed in $O(m \log n)$ time as follows. First we compute set $\mathcal{E}(b)$ for each leaf node b of $\mathcal{T}$. This takes $O(m)$ time. Now $\mathcal{E}(b)$ for an internal node b can be computed by simply merging the lists $\mathcal{E}(b_1)$, $\mathcal{E}(b_2)$ where b_1 and b_2 are children of b. The space complexity of $\mathcal{D}$ is also $O(m \log n)$.

Theorem 2.3. *Given a graph G, it can be preprocessed in $O(m \log n)$ time to build a data structure of size $O(m \log n)$ to perform the following operations.*

1. ACTIVEOUTNGHBRS*($\mathcal{D}, T(w)$): return the set of active nodes in* OUT($T(w)$).
2. MARKINACTIVE*($\mathcal{D}, S$): mark the vertices in set S as inactive.*

The time taken by both of the above operations is $O(deg(S) \log n)$ where S is the set of vertices reported in the first case, and S is the set of vertices marked inactive in the second case.

2.5 Computation of Dominators from Semidominators

In order to find the dominators of the vertices of G it suffices to compute IDOM(v) for each $v \in V$. Our algorithm for computing immediate dominators from semidominators is almost the same as for the restricted case when T is a DFS tree. For the sake of completeness, we now provide this algorithm. The starting point is the concept of relative dominators defined as follows.

Definition 2.8 ([BGK$^+$08])**.** *A vertex w is said to be a relative dominator of v, denoted by $w = rdom(v)$, if w is a descendant of* SDOM(v) *on* PATH$_T$(SDOM(v), v) *for which* SDOM(w) *has the minimum pre-order numbering.*

The following relationship between relative dominators, immediate dominators, and semidominators was shown by Buchbaum et al. [BGK$^+$08] for DFS tree. We show that this relation holds for any arbitrary tree as well.

Lemma 2.4. *Let w be any relative dominator of v. If $w = v$, then* $\text{IDOM}(v) = \text{SDOM}(v)$*, otherwise* $\text{IDOM}(v) = \text{IDOM}(w)$.

Proof. We first consider the case when $w = v$. In order to prove that $\text{SDOM}(v) = \text{IDOM}(v)$, we need to show that (i) $\text{SDOM}(v)$ is a dominator of v, and (ii) none of the internal vertices of $\text{PATH}_T(\text{SDOM}(v), v)$ is a dominator of v. Let us assume on the contrary that there is a path from s to v in $G\backslash\{\text{SDOM}(v)\}$. Then there must exist a detour D starting from an ancestor, say a, of $\text{SDOM}(v)$ and terminating to a descendant, say b, of $\text{SDOM}(v)$ belonging to $\text{PATH}_T(\text{SDOM}(v), v)$. Since existence of detour D implies a valid sequence from a to b, it follows that $\text{SDOM}(b) \in \text{PATH}_T(s, a)$. This contradicts the fact that w is a relative dominator of v. Hence v is unreachable from s in $G\backslash\{\text{SDOM}(v)\}$. Thus in this case $\text{SDOM}(v)$ is a dominator of v. Next let us assume on the contrary that an internal vertex on $\text{PATH}_T(\text{SDOM}(v), v)$, say x, is a dominator of v. Then, $\text{SDOM}(v)$ cannot be parent of v, and so Theorem 2.1 implies there exist two vertex disjoint paths from $\text{SDOM}(v)$ to v. But this contradicts the existence of vertex x, and thus $\text{SDOM}(v)$ must be the immediate dominator of v.

We now consider the case $w \neq v$. Let u be the immediate dominator of w. Again, in order to prove that $u = \text{IDOM}(v)$, we need to show that (i) u is a dominator of v, and (ii) none of the internal vertices of $\text{PATH}_T(u, v)$ is a dominator of v. Let us assume on the contrary that there is a path from s to v in $G\backslash\{u\}$. Since in the graph $G\backslash\{u\}$, w is unreachable from s, there must exist a detour D starting from an ancestor, say a, of u and terminating to a descendant, say b, of w belonging to $\text{PATH}_T(w, v)$. As existence of detour D implies a valid sequence from a to b, it follows that $\text{SDOM}(b) \in \text{PATH}_T(s, a)$. Since a is an ancestor of u, this contradicts the fact that w is a relative dominator of v. Hence v is unreachable from s in $G\backslash\{u\}$. Thus in this case u is a dominator of v. Next let us assume on the contrary that an internal vertex on $\text{PATH}_T(u, v)$, say x, is a dominator of v. Since w is an internal vertex of $\text{PATH}_T(\text{SDOM}(v), v)$, so $\text{SDOM}(v) \neq parent_T(v)$, and thus Theorem 2.1 implies that there exist two vertex disjoint paths from $\text{SDOM}(v)$ to v. This shows that x must lie on $\text{PATH}_T(u, \text{SDOM}(v))$. But there must exist a path from s to w in $G\backslash\{x\}$, say P, as x is not a dominator of v. So P::$\text{PATH}_T(w, v)$ gives a path from s to v in $G\backslash\{x\}$. This contradicts the existence of x, and thus u must be the immediate dominator of v. □

Lemma 2.4 suggests that once we have computed relative dominators, the immediate domina-

tors can be computed in $O(n)$ time by processing the vertices of T in a top down manner. The task of computing relative dominators can be formulated as a data structure problem on a rooted tree as follows.

Each tree edge (u, y) is assigned a weight equal to $\text{SDOM}(y)$. It can be seen that if (a, w) is minimum weight edge on $\text{PATH}_T(\text{SDOM}(v), v)$, then w is a relative dominator of v. So in order to compute relative dominators, all we need is to compute the least weight edge on any given path of tree T. This problem turns out to be an instance of *Bottleneck Edge Query* (BEQ) problem on trees with integral weights. Demaine et al. [DLW14] recently presented the following optimal solution for this problem.

Theorem 2.4 (Demaine et al. [DLW14])**.** *A tree on n vertices and edge weights in the range $[0, n^3]$ can be preprocessed in $O(n)$ time to build a data structure of $O(n)$ size so that given any $u, v \in V$, the edge of smallest weight on $\text{PATH}_T(u, v)$ can be reported in $O(1)$ time.*

We process tree T in a top down order to compute $\text{IDOM}(v)$ as follows. We first compute $rdom(v)$ in $O(1)$ time by performing BEQ query between v and $\text{SDOM}(v)$. Using the data structure stated in Theorem 2.4, it takes $O(1)$ time. Let $w = rdom(v)$. If $w = v$, then we set $\text{IDOM}(v) \leftarrow \text{SDOM}(v)$. Otherwise, we set $\text{IDOM}(v) \leftarrow \text{IDOM}(w)$. Since we process the vertices in a top down fashion, $\text{IDOM}(w)$ has already been computed. Hence it takes $O(1)$ time to compute $\text{IDOM}(v)$. So it can be concluded that we can compute immediate dominators of all vertices in $O(n)$ time only if we know semidominators of all vertices.

Chapter 3

Dual Fault Tolerant Reachability Oracle

In this chapter, we address the problem of building efficient data structures for answering reachability queries from a designated source upon more than one vertex failures. Till now efficient fault tolerant reachability data structures existed only for a single failure using the dominator tree [LT79]. Through a dominator tree one gets an $O(n)$ space data structure that can answer reachability queries after any single failure in $O(1)$ time. We extend this result to dual failures. Our results are summarized as follows:

1. *Oracle:* There exists a data structure of $O(n)$ size that given any two failing vertices f_1, f_2 and a query vertex v, takes $O(1)$ time to determine if v is reachable from s in $G\backslash\{f_1, f_2\}$.

2. *Labeling Scheme:* There exists a compact labeling scheme for answering reachability queries under two failures. Each vertex stores a label of $O(\log^3 n)$ bits such that for any two failing vertices f_1, f_2 and any destination vertex v, it is possible to determine whether v is reachable from s in $G\backslash\{f_1, f_2\}$ by processing the labels associated with f_1, f_2 and v only.

Our result also implies a data structure for the closely related problem of double dominator verification. A pair of vertices (x, y) is said to be *double-dominator* of a vertex v if each path from s to v contains either x or y, but none of x and y are dominators of v. Using our data structure together with the dominator-tree of Lengauer and Tarjan [LT79], one can obtain an $O(n)$ space data structure that for any given triplet $x, y, v \in V$ verifies in $O(1)$ time if (x, y) is double-dominator of v. The best previously known result for this could verify double-dominators only for a fixed s, v pair in $O(1)$ time using an $O(n)$ space data structure called dominator chain [TD05].

3.1 Preliminaries

Throughout this chapter, we use f_1 and f_2 to denote the pair of failed vertices. We use the concept of independent spanning trees which was introduced by Georgiadis and Tarjan [GT12].

Definition 3.1 (Georgiadis and Tarjan [GT12])**.** *Given a directed graph G and a designated source s, a pair of trees, denoted by T_1, T_2, rooted at s are said to be* independent spanning trees *if for each $v \neq s$ the paths from s to v in T_1 and T_2 intersect only at the dominators of v.*

Below we state a few basic properties of dominators in a directed graph.

Property 3.1. *Let T be a reachability tree rooted at s, and y_0, y_1 be vertices such that $y_0 =$ $\text{IDOM}(y_1)$. Then for any $z \in \text{PATH}_T(\bar{y}_0, y_1)$, $\text{IDOM}(z)$ belongs to $\text{PATH}_T(y_0, y_1)$.*

Property 3.2. *Let T be a reachability tree rooted at s, and y_1, y_2 be vertices such that y_1 is ancestor of y_2, and $\text{IDOM}(y_1) = \text{IDOM}(y_2)$. Then for any $z \in \text{PATH}_T(y_1, y_2)$ either y_1 is a dominator of z or $\text{IDOM}(z) = \text{IDOM}(y_1)$.*

For efficient implementation of our oracle, we use the following lemma which is immediate from Theorem 2.4.

Lemma 3.1. *Given a tree, say T, on n vertices, with each vertex assigned an integral weight in range $[0, n^3]$, we can obtain in $O(n)$ time an $O(n)$ size data structure that for any two vertices x, y, outputs in $O(1)$ time the vertex with minimum weight on $\text{PATH}_T(\bar{x}, y)$.*

3.1.1 A heavy path decomposition

The heavy path decomposition of a tree was designed by Sleator and Tarjan [ST83] in the context of dynamic trees. This decomposition has been used in a variety of applications since then. Given any rooted tree T, this decomposition splits T into a set $\mathcal{P}$ of vertex disjoint paths with the property that any path from the root to a leaf node in T can be expressed as a concatenation of at most $\log n$ subpaths of paths in $\mathcal{P}$. This decomposition is carried out as follows. Starting from the root, we follow the path downward such that once we are at a node, say v, the next node traversed is the child of v in T whose subtree is of maximum size, where the size of a subtree is the number of nodes it contains. We terminate upon reaching a leaf node. Let P be the path obtained in this

manner. If we remove P from T, we are left with a collection of subtrees each of size at most $n/2$. Each of these trees hang from P through an edge in T. We carry out the decomposition of these trees recursively. The following lemma is immediate from the construction of a heavy path decomposition.

Lemma 3.2. *Let T be any rooted tree of G and $\mathcal{P}$ be the collection of paths obtained from a heavy path decomposition of T. Then for any vertex $v \in V$, the number of paths in $\mathcal{P}$ which start from either v or an ancestor of v in T is at most* $\log n$.

3.2 Detour based Reachability Oracle for handling Single Failure

In order to understand the dual fault tolerant reachability oracle we first briefly discuss the case of single failure. We here describe an alternative reachability oracle using detours instead of dominator tree. Let T be any reachability tree of G rooted at s, and f, v be respectively the failed and the query vertex. Also assume f is an ancestor of v in T. Notice that if v is reachable from s in $G \setminus \{f\}$, then there must exist a path starting from $\text{PATH}_T(s, \bar{f})$ and terminating at $\text{PATH}_T(\bar{f}, v)$ which, except for its endpoints, does not pass through any ancestor of v in T. So for each $w \in V$, we can define a detour $D(w)$ to be a path starting from the highest possible ancestor of w in T and terminating at w such that none of the internal vertices of the path pass through an ancestor of w. Now on failure of f it suffices to search whether there exists a vertex lying in $\text{PATH}_T(\bar{f}, v)$ whose detour starts from an ancestor of f. This can be achieved by assigning to each vertex w a weight equal to the depth of the first vertex on $D(w)$. By doing this the problem of reachability under one vertex failure reduces to the problem of solving range minima on weighted trees, for which already an optimal solution exists. (See Lemma 3.1).

3.3 Overview

Let us consider the failure of a pair of vertices f_1, f_2 in G, and let v be the query vertex. Note that if any of the tree paths - $\text{PATH}_{T_1}(s, v)$ or $\text{PATH}_{T_2}(s, v)$ is intact, then v will be reachable from s. Also, if both $\text{PATH}_{T_1}(s, v)$ or $\text{PATH}_{T_2}(s, v)$ contains a common failed vertex, say f_1, then v will not be reachable from s. This is because then f_1 would be a dominator of v. Thus the non-trivial

case is when $\text{PATH}_{T_1}(s,v)$ contains only f_1 and $\text{PATH}_{T_2}(s,v)$ contains only f_2, or the vice-versa. So whenever a query vertex v is given to us, we may assume that the following condition is satisfied.

$\mathcal{C}$: f_1 lies on $\text{PATH}_{T_1}(s,v) \backslash \text{PATH}_{T_2}(s,v)$, and f_2 lies on $\text{PATH}_{T_2}(s,v) \backslash \text{PATH}_{T_1}(s,v)$.

Now consider the sets $S_A(v)$ and $S_B(v)$ as defined below. (For a better understanding of these sets see Figure 3.1(i)).

- $S_A(v)$: Set of vertices lying either above f_1 on $\text{PATH}_{T_1}(s,v)$ or above f_2 on $\text{PATH}_{T_2}(s,v)$.
- $S_B(v)$: Set of vertices lying either below f_1 on $\text{PATH}_{T_1}(s,v)$ or below f_2 on $\text{PATH}_{T_2}(s,v)$.

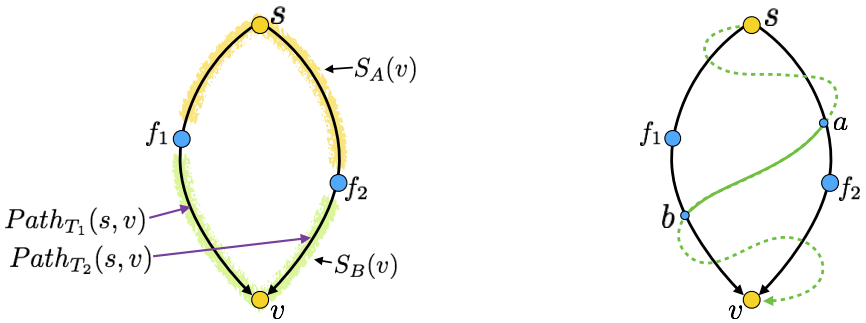


Figure 3.1: (i) Representation of sets $S_A(v)$ and $S_B(v)$ when condition $\mathcal{C}$ is satisfied for vertex v; (ii) A path from $a \in S_A(v)$ to $b \in S_B(v)$ when v is reachable from s in $G \backslash \{f_1, f_2\}$.

It turns out that if v is reachable from s in $G \backslash \{f_1, f_2\}$, then there must exists a path from set $S_A(v)$ to $S_B(v)$ avoiding the vertices of both $\text{PATH}_{T_1}(s,v)$ and $\text{PATH}_{T_2}(s,v)$. This fact is formally stated in the following lemma.

Lemma 3.3. *Given a pair of failed vertices f_1, f_2, a vertex v is reachable from s if and only if G contains a path P satisfying the following conditions. (See Figure 3.1(ii)).*

C1. The first and last vertices of P lies respectively in sets $S_A(v)$ and $S_B(v)$.

C2. None of the internal vertices of P lies on $\text{PATH}_{T_1}(s,v)$ or $\text{PATH}_{T_2}(s,v)$.

Proof. We first consider the case that v is reachable from s after the failure of f_1, f_2. Let Q be any path from s to v in $G \backslash \{f_1, f_2\}$. Let a be the last vertex on Q lying in the set $S_A(v)$,

and b be the first vertex on $Q[a, v]$ that lies in the set $S_B(v)$. The vertices a, b are well defined since $s \in S_A(v)$ and $v \in S_B(v)$. It is easy to see that none of the intermediate vertices of path $Q[a, b]$ lies on $\text{PATH}_{T_1}(s, v)$ and $\text{PATH}_{T_2}(s, v)$. Hence $Q[a, b]$ is the required path which starts from a vertex in $S_A(v)$ and terminates to a vertex in $S_B(v)$. Now to prove the converse let us assume that there exists a path, say P, from $a \in S_A(v)$ to $b \in S_B(v)$ such that none of the intermediate vertices of P lies on $\text{PATH}_{T_1}(s, v)$ and $\text{PATH}_{T_2}(s, v)$. Let $i, j \in [1, 2]$ be such that $a \in \text{PATH}_{T_i}(s, f_i)$ and $b \in \text{PATH}_{T_j}(f_j, v)$. Note that the paths $\text{PATH}_{T_i}(s, a)$, $\text{PATH}_{T_j}(b, v)$, and P cannot contain the failed vertices f_1 or f_2. Hence $\text{PATH}_{T_i}(s, a)::P::\text{PATH}_{T_j}(b, v)$ is a path from s to v in $G\backslash\{f_1, f_2\}$. □

For simplicity we refer to a path satisfying the conditions *C1* and *C2* stated in the above lemma as an $S_{A,B}(v)$ path. In order to efficiently compute such a path we define a pair of detours $D^1(w)$ and $D^2(w)$ for each vertex $w \in V$ as follows.

- $\text{D}^i(w)$: a path starting from the highest possible ancestor of w in T_i and terminating at w such that none of the internal vertices of the path are ancestor of w in T_1 or T_2.

Note that the detours $D^1(w)$ and $D^2(w)$ can be seen as a simple generalization of the detours defined in Chapter 2 (see Definition 2.3). However, we show that this simple generalization is not sufficient to answer the reachability queries in dual failure. To understand this subtle point consider an $S_{A,B}(v)$ path with a, b as its endpoints. If the endpoint b is equal to v, then P could be simply either $D^1(v)$ or $D^2(v)$. The problem arises when $b \neq v$. This is because if b is an ancestor of v in T_1, then P might contain vertices from $\text{PATH}_{T_2}(s, b)$. (Recall that the internal vertices of P are disjoint from $\text{PATH}_{T_2}(s, v)$, but not necessarily disjoint from $\text{PATH}_{T_2}(s, b)$). So in this case P can neither be $D^1(b)$ nor be $D^2(b)$. For a more clear insight into this consider the graph and its two independent spanning trees in Figure 3.2. Since in-degree of each vertex in the graph is at most two, $\text{D}^i(w)$ for each $w \in V$ is simply the incoming edge from $parent_{T_i}(w)$ to w. Thus the path P connecting $S_A(v)$ to $S_B(v)$ is a concatenation of as many detours as there are number of edges in P. Determining whether a concatenation of all these single-edge detours can give us an $S_{A,B}(v)$ path is difficult to achieve in $O(1)$ time.

This shows that a simple generalization of detours from a single tree to two trees is not sufficient. To tackle the problem we extend the notion of detours to 'Parent Detours' and 'Ancestor

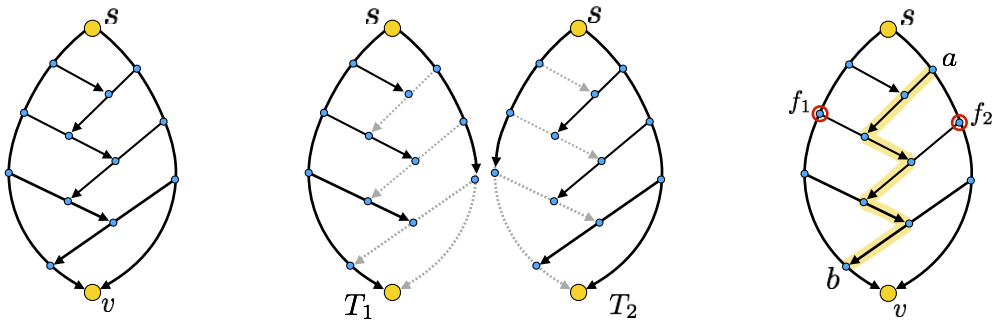


Figure 3.2: (i) A graph G with in-degree of each vertex bounded by two; (ii) A pair of independent spanning trees T_1 and T_2 for G; (iii) A path P from $a \in S_A(v)$ to $b \in S_B(v)$ in $G \backslash \{f_1, f_2\}$.

Detours'. These detours unlike the normal detour terminates at an appropriate ancestor of w in T_1 or T_2. We formally define the parent-detours and ancestor-detours in the following sections, and show how they can be used to solve the problem of dual fault tolerant reachability.

3.4 Reachability Oracle for 2-Vertex Strongly Connected Graphs

In this section we describe an $O(n)$ space and $O(1)$ time reachability oracle for 2-vertex strongly connected graphs. By 2-vertex strongly connectedness we have that on removal of any vertex f ($f \neq s$), all the vertices in $G \backslash \{f\}$ are still reachable from s. Thus each vertex is dominated only by source s and by itself. This implies that for any vertex w, $\text{PATH}_{T_1}(s, w)$ and $\text{PATH}_{T_2}(s, w)$ intersects only at the endpoints s, w.

Consider a query vertex v which is reachable from s in $G \backslash \{f_1, f_2\}$. Let us assume that condition $\mathcal{C}$ is satisfied for v. Let P be any $S_{A,B}(v)$ path, and a, b be respectively the first and last vertices on P. Without loss of generality we can assume that b lies on $\text{PATH}_{T_1}(s, v)$. See Figure 3.3(i). We make the following additional assumptions.

1. None of the $S_{A,B}(v)$ paths terminates at v (i.e. b cannot be v).

2. b is the lowest vertex on $\text{PATH}_{T_1}(s, v)$ at which an $S_{A,B}(v)$ path terminates.

Remark 3.1. *The assumption 1 is justified since if $b = v$, then v will be reachable from s using the detours $D^1(v)$ or $D^2(v)$.*

We now state a lemma which provides the motivation for defining the parent detours.

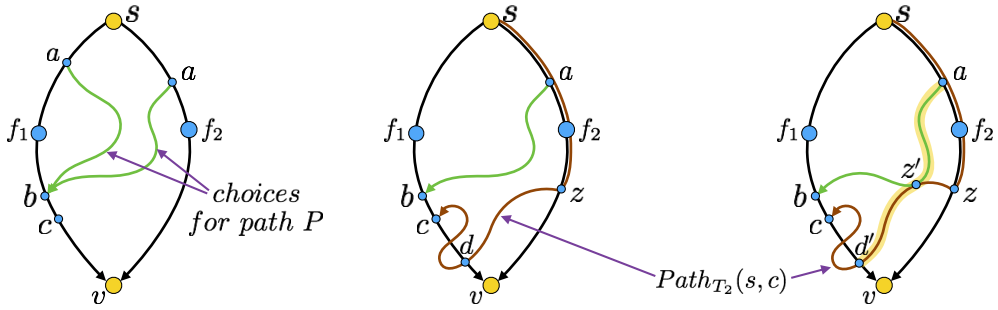
Lemma 3.4. *Let a, b, P be as described above, and c be the child of b on* $\text{PATH}_{T_1}(s, v)$. *Then,*

(i) Vertex f_2 is an ancestor of c in T_2.

(ii) None of the internal vertices of P lie on $\text{PATH}_{T_1}(s, c)$ *or* $\text{PATH}_{T_2}(s, c)$.

Proof. Let z denote the LCA of vertices c and v in tree T_2. (See Figure 3.3(ii)). Consider the path $Q = \text{PATH}_{T_2}(z, c)$. It is easy to see that none of the internal vertices of Q lies on $\text{PATH}_{T_2}(s, v)$. Also, the internal vertices of Q appearing on $\text{PATH}_{T_1}(s, v)$ must lie below c on $\text{PATH}_{T_1}(s, v)$. This is because, by definition of independent spanning trees, $\text{PATH}_{T_1}(s, c)$ and Q can intersect only at the vertices s and c.

We now prove claim 1. Let d be the first point of intersection of Q with $\text{PATH}_{T_1}(c, v)$. (See Figure 3.3(ii)). Then the internal vertices of $Q[z, d]$ are disjoint from both $\text{PATH}_{T_1}(s, v)$ and $\text{PATH}_{T_2}(s, v)$. Now if z is an ancestor of f_2 in T_2, then $Q[z, d]$ forms an $S_{A,B}(v)$ path, terminating at descendant of b in T_1, thereby violating assumption 2. Hence f_2 must be either same as z or an ancestor of z. This shows that f_2 is an ancestor of c in T_2.



(i) (ii) (iii)

Figure 3.3: (i) Possibilities for path P when $b \in \text{PATH}_{T_1}(\bar{f}_1, \bar{v})$; (ii) Representation of $\text{PATH}_{T_2}(s, c)$ and $z = LCA(c, v)$ in T_2; (iii) Violation of assumption 2 if $P \cap \text{PATH}_{T_2}(z, c)$ is non-empty.

In order to prove claim 2, we first show that P is disjoint from Q. Let us suppose on the contrary, that there exists a vertex, say z', belonging to $P \cap Q$. Also let d' be the first vertex of $Q[z', c]$ lying on $\text{PATH}_{T_1}(c, v)$. (See Figure 3.3(iii)). Then $P[a, z']::Q[z', d']$ forms an $S_{A,B}(v)$ path terminating at a descendant of b in T_1. This again violates assumption 2. Thus $P \cap Q = \emptyset$. Now since the internal vertices of P are disjoint from $\text{PATH}_{T_1}(s, v), \text{PATH}_{T_2}(s, v)$, they must be disjoint from $\text{PATH}_{T_1}(s, c)$ and $\text{PATH}_{T_2}(s, z)::Q = \text{PATH}_{T_2}(s, c)$, as well. □

The above lemma implies that f_1 is an ancestor of c in T_1, and f_2 is an ancestor of c in T_2.

Thus $S_A(v) = S_A(c)$. Hence we have the following corollary.

Corollary 3.1. *P is an $S_{A,B}(c)$ path terminating at $parent_{T_1}(c)$.*

In order to capture the above fact we define parent-detours for each $w \in V$ which instead of terminating at w terminates at either $parent_{T_1}(w)$ or $parent_{T_2}(w)$.

- $\text{PD}^i_j(w)$: a path starting from the highest possible ancestor of w in T_i and terminating at $parent_{T_j}(w)$ s.t. none of the internal vertices of the path lie on $\text{PATH}_{T_1}(s, w)$ or $\text{PATH}_{T_2}(s, w)$.

By above definition of parent-detour it follows that P can be replaced by either $\text{PD}^1_1(c)$ or $\text{PD}^2_1(c)$ depending upon whether it starts from an ancestor of c in T_1 or T_2. Now let x_1 denote the child of f_1 in T_1 lying on $\text{PATH}_{T_1}(s, v)$, and x_2 denote the child of f_2 in T_2 lying on $\text{PATH}_{T_2}(s, v)$. Then the parent-detours of vertices x_1, x_2 may not be of any help, since they would terminate at f_1 and f_2. However, the parent-detours of vertices in $S_B(v)\backslash\{x_1, x_2\}$ will suffice to determine whether v is reachable from s or not.

In above discussion, we observed that P is an $S_{A,B}(c)$ path terminating at $b = parent_{T_1}(c)$. This shows that for vertices lying on $\text{PATH}_{T_1}(\bar{x}_1, v)$, we only need to worry about parent-detours terminating at $parent_{T_1}(\cdot)$, i.e. $\text{PD}^1_1(\cdot)$ and $\text{PD}^2_1(\cdot)$. Whereas, for vertices on $\text{PATH}_{T_2}(\bar{x}_2, v)$, we need to worry about parent-detours terminating at $parent_{T_2}(\cdot)$, i.e. $\text{PD}^1_2(\cdot)$ and $\text{PD}^2_2(\cdot)$. We thus have the following lemma.

Lemma 3.5. *Let x_1 and x_2 be as defined above. A vertex v is reachable from s in $G\backslash\{f_1, f_2\}$ if and only if at least one of the following vertices lie in $S_A(v)$.*

(i) The first vertex of $\text{D}^1(v)$ or $\text{D}^2(v)$.

(ii) The first vertex of either $\text{PD}^1_1(w)$ or $\text{PD}^2_1(w)$ for some $w \in \text{PATH}_{T_1}(\bar{x}_1, v)$.

(iii) The first vertex of either $\text{PD}^1_2(w)$ or $\text{PD}^2_2(w)$ for some $w \in \text{PATH}_{T_2}(\bar{x}_2, v)$.

3.4.1 Implementation of the Oracle

We first introduce the following notations for detours and parent detours.

- $\beta^i(v)$: $depth_{T_i}$(first vertex on $\text{D}^i(v)$).

- $\gamma_j^i(v)$: $depth_{T_i}$(first vertex on $\text{PD}_j^i(v)$).

Now let f_1, f_2 be a given pair of failed vertices and v be a given query vertex. Our first step is to check if condition $\mathcal{C}$ is satisfied. Recall that this requires only verifying the ancestor-descendant relationship in trees T_1 and T_2. One simple method to achieve this for any given tree T is to perform the pre-order and the post-order traversal of T, and store the vertices in the order they are visited. Now x will be ancestor of y in T if and only if x appears before y in the pre-order traversal, and after y in the post-order traversal.

Algorithm 3.1 presents the pseudo-code for answering reachability query for a vertex v assuming condition $\mathcal{C}$ is satisfied. This can be explained in words as follows. For $i = 1, 2$, we first check if $D^i(v)$ starts from an ancestor of f_i in T_i or not. This is done by comparing the value of $\beta^i(v)$ with the depth of f_i in T_i. Next we compute the vertices x_1, x_2. Finally for $i, j \in \{1, 2\}$, we compute a vertex $w \in \text{PATH}_{T_j}(\bar{x}_j, v)$ for which $\gamma_j^i(\cdot)$ is minimum. If $\gamma_j^i(w)$ is less than the depth of f_i in T_i, then it implies that $\text{PD}_j^i(w)$ starts from an ancestor of f_i in T_i, so we return True. If we reach to the end of code, that means we have not been able to find any path for v, so we return False.

Algorithm 3.1: Oracle for reachability to v in 2-vertex strongly connected graphs.

1 **if** $\beta^1(v) < depth_{T_1}(f_1)$ *or* $\beta^2(v) < depth_{T_2}(f_2)$ **then** Return True;
2 $x_1 \leftarrow$ the vertex with minimum depth on $\text{PATH}_{T_1}(\bar{f}_1, v)$;
3 $x_2 \leftarrow$ the vertex with minimum depth on $\text{PATH}_{T_2}(\bar{f}_2, v)$;
4 **foreach** $i, j \in \{1, 2\}$ **do**
5 $\quad w \leftarrow$ a vertex on $\text{PATH}_{T_j}(\bar{x}_j, v)$ for which $\gamma_j^i(\cdot)$ is minimum;
6 $\quad$ **if** $\gamma_j^i(w) < depth_{T_i}(f_i)$ **then** Return True;
7 **end**
8 Return False;

The above oracle can be easily implemented in $O(1)$ time, by having a total of six weight functions - one each for storing the depth of a vertex in trees T_1, T_2, and the other four for storing the values $\gamma_j^i(\cdot)$, for $i, j \in \{1, 2\}$. By doing this the vertices x_1, x_2 can be computed in constant time since they are respectively the vertices with minimum depth on the paths $\text{PATH}_{T_1}(\bar{f}_1, v)$ and $\text{PATH}_{T_1}(\bar{f}_1, v)$. Also Step 4 can be carried out in $O(1)$ time. Next notice that the independent spanning trees T_1 and T_2 in Definition 3.1, and the data structure of Lemma 3.1 can respectively be computed in $O(m)$ and $O(n)$ time. Also for any vertex v, the detours $\text{D}^i(v)$ and $\text{PD}_j^i(v)$ can be

computed in $O(m)$ time. So, it is easy to see that the preprocessing time of our oracle is $O(mn)$. We can thus state the following theorem.

Theorem 3.1. *A 2-vertex strongly connected graph on n vertices can be preprocessed in $O(mn)$ time for a given source vertex s to build a data structure of $O(n)$ size such that for any query vertex v, and pair of failures f_1, f_2, it takes $O(1)$ time to determine if there exists any path from s to v in $G\backslash\{f_1, f_2\}$.*

3.5 Reachability Oracle for General Graphs

In this section we explain the reachability oracle for general graphs. Consider a query vertex u in G. Let $u_0, u_1, ..., u_k$ be the dominators of u with $u_0 = s$ and $u_k = u$. Thus $\text{PATH}_{T_1}(s, u)$ and $\text{PATH}_{T_2}(s, u)$ intersect only at u_i's. (See Figure 3.4(i)). As in Section 3.4, we assume that condition $\mathcal{C}$ holds for u, so none of the u_i's can be equal to f_1 or f_2. Now let $i, j \in [1, k]$ be such that $f_1 \in \text{PATH}_{T_1}(\bar{u}_{i-1}, \bar{u}_i)$ and $f_2 \in \text{PATH}_{T_2}(\bar{u}_{j-1}, \bar{u}_j)$. It is easy to see that if $i \neq j$, then u is reachable from s by the path $\text{PATH}_{T_1}(s, u_{i-1})$::$\text{PATH}_{T_2}(u_{i-1}, u_i)$::$\text{PATH}_{T_1}(u_i, u)$. (See Figure 3.4(ii)). Thus we consider the case when $i = j$ [1]. For simplicity, we use symbols, v and $\text{IDOM}(v)$ to respectively denote the vertices u_i and u_{i-1}. Notice that in order to check reachability of u from s, it suffices to check if v is reachable from s in $G\backslash\{f_1, f_2\}$.

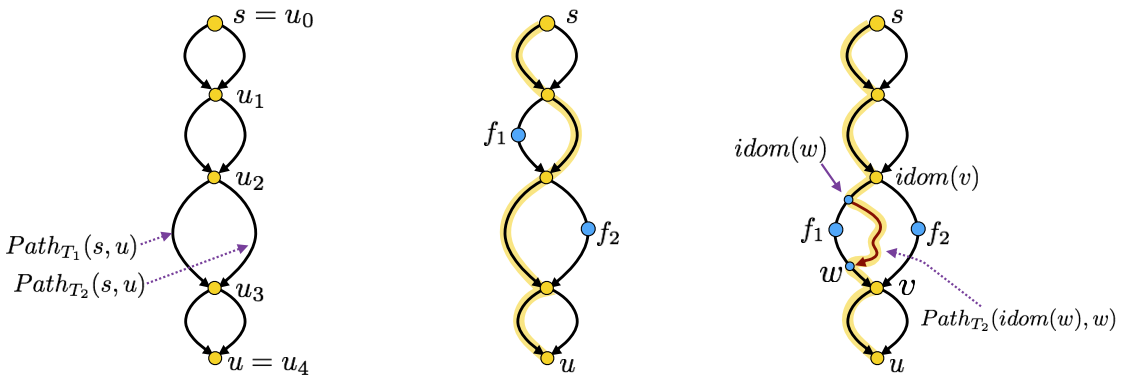


Figure 3.4: (i) Representation of dominators of u; (ii) A path from s to u (highlighted in yellow) when f_1 lies on $\text{PATH}_{T_1}(\bar{u}_1, \bar{u}_2)$ and f_2 lies on $\text{PATH}_{T_2}(\bar{u}_2, \bar{u}_3)$; (iii) A path from s to u (highlighted in yellow) when there exists a vertex $w \in S_B(v)$ for which $\text{IDOM}(w)$ lies in the set $S_A(v)\backslash\{\text{IDOM}(v)\}$.

[1] One can verify in $O(1)$ time whether $u_{i-1} = u_{j-1}$ (i.e. if $i = j$) since vertex $u_{i-1} = LCA(f_1, v)$ and vertex $u_{j-1} = LCA(f_2, v)$ in the dominator tree of G.

We now divide our analysis into various different cases as follows:

Case 1. *There exists an $S_{A,B}(v)$ path terminating at vertex v.*

In this case v will be reachable from s using either of the detours $D^1(v)$ or $D^2(v)$.

Case 2. *There exists a vertex $w \in S_B(v)$ for which* $\text{IDOM}(w) \in S_A(v) \backslash \{\text{IDOM}(v)\}$.

In this case also we can show that v is reachable from s by the following argument. Without loss of generality let us assume that w is an ancestor of v in T_1. Since $\text{IDOM}(w)$ is an ancestor of w in T_1, it must lie on $\text{PATH}_{T_1}(\overline{\text{IDOM}(v)}, \bar{f}_1)$. (See Figure 3.4(iii)). Consider the path $Q = \text{PATH}_{T_2}(\text{IDOM}(w), w)$. Note that f_2 cannot lie on Q. This is because otherwise $\text{PATH}_{T_1}(\text{IDOM}(v), w)$ and $\text{PATH}_{T_2}(\text{IDOM}(v), f_2)::Q[f_2, w]$ will form two vertex disjoint paths from $\text{IDOM}(v)$ to w, which would violate the fact that $\text{IDOM}(w) \neq \text{IDOM}(v)$. Also f_1 cannot lie on Q, as Q is disjoint from $\text{PATH}_{T_1}(\overline{\text{IDOM}(w)}, \bar{w})$. Thus v is reachable from s by the path $\text{PATH}_{T_1}(s, \text{IDOM}(w))::Q::\text{PATH}_{T_1}(w, v)$.

Case 3. *None of the $S_{A,B}(v)$ path terminates at v, and there does not exist a vertex in $S_B(v)$ whose immediate dominator lies in* $S_A(v) \backslash \{\text{IDOM}(v)\}$.

This is the most non-trivial case of dual fault tolerant reachability oracle. We now provide analysis for this case.

Let us suppose v is reachable from s in $G \backslash \{f_1, f_2\}$. Then without loss of generality we can assume that there exists an $S_{A,B}(v)$ path (say P) terminating at an ancestor of v in T_1. In case there are multiple $S_{A,B}(v)$ paths, then we take P to be that path which terminates at lowest vertex on $\text{PATH}_{T_1}(\bar{f}_1, \bar{v})$. Let a, b be respectively the first and last vertices on P. By Property 3.1, we know that $\text{IDOM}(b)$ cannot be an ancestor of $\text{IDOM}(v)$ in T_1. Therefore, $\text{IDOM}(b)$ must be equal to $\text{IDOM}(v)$. This is because $\text{IDOM}(b)$ cannot lie in $S_A(v) \backslash \{\text{IDOM}(v)\}$, and if $\text{IDOM}(b)$ lies in $S_B(v)$ then $P \cap S_B(v)$ will contain both b and $\text{IDOM}(b)$, which would violate the definition of an $S_{A,B}(v)$ path.

Now consider the vertex c which is child of b on $\text{PATH}_{T_1}(s, v)$. It turns out that in a general graph the parent-detours of c may not be of any help. This is because the analysis for 2-vertex strongly connected graphs crucially exploited the fact that $\text{IDOM}(b) = \text{IDOM}(c) = s$. But in general graphs, if $\text{IDOM}(b)$ is not equal to $\text{IDOM}(c)$, then it can be shown that Lemma 3.4 no

longer holds. To be more precise, we can show that the internal vertices of P might not be disjoint from $\text{PATH}_{T_2}(s, c)$.

However, the problem can be resolved if we take c to be the first descendant of b on $\text{PATH}_{T_1}(s, v)$ whose immediate dominator is the same as that of b. This motivates us to define the notion of pseudo-child (and pseudo-parent) as follows.

Definition 3.2. *Given a reachability tree T rooted at s, a vertex x is said to be pseudo-parent of y in T (and y is said to be pseudo-child of x) if x is the nearest ancestor of y in T whose immediate dominator is the same as that of y.*

Note that in a 2-vertex strongly connected graph, the definition of pseudo-parent and pseudo-child degenerates to normal notion of parent and child. This is because, immediate dominator of all the vertices (other than s) in such graph is equal to s.

We now state a lemma which is an analogue of Lemma 3.4 for general graphs.

Lemma 3.6. *Let a, b, P be as described above, and c be the pseudo-child of b on $\text{PATH}_{T_1}(s, v)$. Then,*
(i) Vertex f_2 is an ancestor of c in T_2.
(ii) None of the internal vertices of P lie on $\text{PATH}_{T_1}(s, c)$ or $\text{PATH}_{T_2}(s, c)$.

As a corollary of the above lemma we get that P is an $S_{A,B}(c)$ path terminating at pseudo-parent of c in T_1. We thus define ancestor-detours which are a generalization of parent-detours as follows.

- $\text{AD}^i_j(w)$: a path starting from the highest possible ancestor of w in T_i and terminating at pseudo-parent of w in tree T_j such that none of the internal vertices of the path lie on $\text{PATH}_{T_1}(s, w)$ or $\text{PATH}_{T_2}(s, w)$.

Now let x_1 be the first descendant of f_1 on $\text{PATH}_{T_1}(s, v)$ whose immediate dominator is equal to $\text{IDOM}(v)$. Similarly, let x_2 be the first descendant of f_2 on $\text{PATH}_{T_1}(s, v)$ whose immediate dominator is equal to $\text{IDOM}(v)$. Then the ancestor-detours of x_1, x_2 will not be of any help as they would terminate at either f_1, f_2 or their ancestors. Now as in Section 3.4, we can argue that ancestor-detours of vertices on $\text{PATH}_{T_1}(\bar{x}_1, v) \cup \text{PATH}_{T_2}(\bar{x}_2, v)$ suffice to answer the reachability query for vertex v. This completes the analysis of the third case.

We thus have the following lemma.

Lemma 3.7. *Let v be a vertex satisfying condition $\mathcal{C}$ such that $f_1 \in \text{PATH}_{T_1}(\overline{\text{IDOM}(v)}, v)$ and $f_2 \in \text{PATH}_{T_1}(\overline{\text{IDOM}(v)}, v)$. Also let x_1 and x_2 be as defined above. Then v is reachable from s in $G\backslash\{f_1, f_2\}$ if and only if either of the following statements holds true.*

(i) [Case 1] The first vertex of $\text{D}^1(v)$ or $\text{D}^2(v)$ lies in $S_A(v)$.

(ii) [Case 2] There exists a vertex $w \in S_B(v)$ for which $\text{IDOM}(w) \in S_A(v)\backslash\{\text{IDOM}(v)\}$.

(iii) [Case 3] There exists a vertex $w \in \text{PATH}_{T_1}(\bar{x}_1, v)$ such that $\text{IDOM}(w) = \text{IDOM}(v)$ and the first vertex of either $\text{AD}_1^1(w)$ or $\text{AD}_1^2(w)$ lies in $S_A(v)$.

(iv) [Case 3] There exists a vertex $w \in \text{PATH}_{T_2}(\bar{x}_2, v)$ such that $\text{IDOM}(w) = \text{IDOM}(v)$ and the first vertex of either $\text{AD}_2^1(w)$ or $\text{AD}_2^2(w)$ lies in $S_A(v)$.

3.5.1 Implementation of the Oracle

We now explain the implementation of reachability oracle for general graphs. As in Section 3.4, we define the following notations.

- $\alpha^i(v)$: $depth_{T_i}(\text{IDOM}(v))$.
- $\beta^i(v)$: $depth_{T_i}(\text{first vertex on } \text{D}^i(v))$.
- $\gamma_j^i(v)$: $depth_{T_i}(\text{first vertex on } \text{AD}_j^i(v))$.

Let f_1, f_2 be a given pair of failed vertices and v be a given query step. We assume that condition $\mathcal{C}$ is satisfied, and the failures f_1, f_2 lie respectively on the paths $\text{PATH}_{T_1}(\text{IDOM}(v), v)$ and $\text{PATH}_{T_2}(\text{IDOM}(v), v)$. We first check for $i = 1, 2$, if $D^i(v)$ starts from an ancestor of f_i in T_i or not. This is done by comparing the value of $\beta^i(v)$ with the depth of f_i in T_i.

Next we compute the vertices x_1, x_2 as follows. Recall that x_1 is the highest ancestor of v in $\text{PATH}_{T_1}(\bar{f}_1, v)$ whose immediate dominator is equal to $\text{IDOM}(v)$. So to obtain x_1, we call the range minima query for vertices on $\text{PATH}_{T_1}(\bar{f}_1, v)$ with $\langle \alpha^1(\cdot), depth_{T_1}(\cdot) \rangle$ as the weight function. By comparing the value of $\alpha^1(\cdot)$, it is able to filter out those vertices in $\text{PATH}_{T_1}(\bar{f}_1, v)$ whose

immediate dominator is at minimum depth in T_1, i.e. it is equal to $\text{IDOM}(v)$. After this it assigns x_1 to be that vertex which has minimum depth in T_1. Vertex x_2 is computed in a similar manner.

Now notice that to find whether there exists a vertex in $S_B(v)$ whose immediate dominator lies in $S_A(v)\backslash\{\text{IDOM}(v)\}$, we only need to restrict ourself to paths $\text{PATH}_{T_1}(\bar{f}_1, \bar{x}_1)$ and $\text{PATH}_{T_1}(\bar{f}_1, \bar{x}_1)$. This is because Property 3.2 implies that immediate dominator of vertices in $\text{PATH}_{T_1}(x_1, v)$ is either equal to $\text{IDOM}(v)$ or lies in $\text{PATH}_{T_1}(x_1, v)$ itself. Similarly, for $\text{PATH}_{T_2}(x_2, v)$. So for $i = 1, 2$, we perform the range minima query to find a vertex, say w, on $\text{PATH}_{T_i}(\bar{f}_i, \bar{x}_i)$ for which $\alpha^i(w)$ is minimum. If $\alpha^i(w)$ is less than the depth of f_i in T_i, then we report that v is reachable from s.

Finally for $i, j \in \{1, 2\}$, we compute a vertex $w \in \text{PATH}_{T_j}(\bar{x}_j, v)$ for which $\langle \alpha^i(\cdot), \gamma^i_j(\cdot) \rangle$ is minimum. The term $\alpha^i(\cdot)$ is added in front so that we are able to filter out those vertices whose immediate dominator is equal to $\text{IDOM}(v)$. Now if $\gamma^i_j(w)$ is less than the depth of f_i in T_i, then it implies that $\text{AD}^i_j(w)$ starts from an ancestor of f_i in T_i, so we return True.

If we reach to the end of code, that means we have not been able to find any path for v, so we return False.

Algorithm 3.2: Oracle for reachability to v in general graphs.

```
1  if β¹(v) < depth_{T1}(f1) or β²(v) < depth_{T2}(f2) then Return True;
2  x1 ← a vertex on PATH_{T1}(f̄1, v) for which ⟨α¹(·), depth_{T1}(·)⟩ is minimum;
3  x2 ← a vertex on PATH_{T2}(f̄2, v) for which ⟨α²(·), depth_{T2}(·)⟩ is minimum;
4  foreach i ∈ {1, 2} do
5  |   w ← a vertex on PATH_{Ti}(f̄i, x̄i) for which αⁱ(·) is minimum;
6  |   if αⁱ(w) < depth_{Ti}(fi) then Return True;
7  end
8  foreach i, j ∈ {1, 2} do
9  |   w ← a vertex on PATH_{Tj}(x̄j, v) for which ⟨αⁱ(·), γⁱⱼ(·)⟩ is minimum;
10 |   if γⁱⱼ(w) < depth_{Ti}(fi) then Return True;
11 end
12 Return False;
```

As in Algorithm 3.1, we can argue that the Steps 2, 3, 5 and 9 can be implemented in $O(1)$ time. Thus Algorithm 3.2 takes constant time to answer reachability queries. Also, one can see that similar to Section 3.4, the preprocessing time of our oracle is $O(mn)$. We thus conclude with the following theorem.

Theorem 3.2. *A directed graph $G = (V, E)$ on n vertices can be preprocessed in $O(mn)$ time for a given source vertex $s \in V$ to build a data structure of $O(n)$ size such that for any $f_1, f_2, v \in V$,*

it takes $O(1)$ time to determine if there exists a path from s to v in $G\backslash\{f_1, f_2\}$.

3.6 Labeling Scheme

In this section we describe the labeling scheme for answering reachability queries from s to any vertex v on two vertex failures. In the first subsection we develop a labeling scheme for reporting minima and second minima on tree paths. In the next subsection, we see how these labels can be used to obtain a labeling scheme for answering reachability queries.

3.6.1 Labeling Scheme for Minima and Second-Minima on Tree Paths

We assume that a reachability tree T rooted at s is given to us, and each vertex x is assigned a positive integral weight $wt(x)$ in the range $[0, n^2]$. Note that second-minima may have two interpretations depending upon whether or not the second minimum value is allowed to be the same as the first one. In this chapter, by second-minima we always mean the smallest value obtained after removing all occurrence of the first minimum value. The first minimum and second minimum operations are respectively denoted by $\min_1(\cdot)$ and $\min_2(\cdot)$. Now given any two vertices x, y, where x is ancestor of y in T, we define the following notations.

- $\text{FM}(x, y) := \min_1\{wt(z) \mid z \in \text{PATH}_T(\bar{x}, y)\}$.
- $\text{SM}(x, y) := \min_2\{wt(z) \mid z \in \text{PATH}_T(\bar{x}, y)\}$.

Our aim is to develop compact labeling scheme for reporting $\text{FM}(x, y)$ and $\text{SM}(x, y)$ described above. In order to achieve this we first perform a heavy path decomposition of T (refer to Subsection 3.1.1). Let $\mathcal{P}$ be the collection of paths obtained by this decomposition. For each vertex x, we use $P(x)$ to denote the path in $\mathcal{P}$ containing x. Now for each vertex x and each integer $i \in [1, \log n]$, we define the following notations.

- $\text{FM}_a(x, i) := \min_1\{wt(y) \mid y \text{ lies } \underline{\text{a}}\text{bove } x \text{ in } P(x) \text{ and } |depth_T(x) - depth_T(y)| \leq 2^i\}$.
- $\text{SM}_a(x, i) := \min_2\{wt(y) \mid y \text{ lies } \underline{\text{a}}\text{bove } x \text{ in } P(x) \text{ and } |depth_T(x) - depth_T(y)| \leq 2^i\}$.

Similarly, we define the notations $\text{FM}_b(x, i)$ and $\text{SM}_b(x, i)$. For each vertex x, let $\delta(x)$ denote the string of $O(\log^2 n)$ bits obtained by concatenation of: (i) $depth_T(x)$, (ii) $wt(x)$, and (iii) $\text{FM}_a(x, i), \text{SM}_a(x, i), \text{FM}_b(x, i), \text{SM}_b(x, i)$ for each $i \in [1, \log n]$.

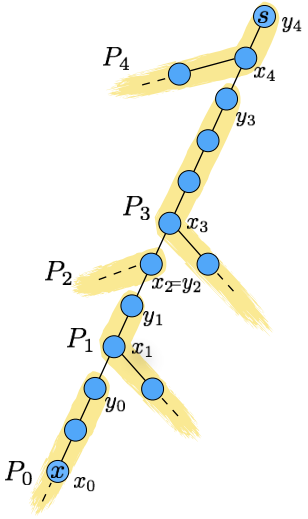


Figure 3.5: Representation of paths $P_0, P_1, .., P_t$ where $P_0 = P(x)$ and $P_1, .., P_t$ are paths in $\mathcal{P}$ starting from an ancestor of x. Note that t will be bounded by $\log n$.

- $\delta(x) := \langle depth_T(x), wt(x), \text{FM}_a(x,i), \text{SM}_a(x,i), \text{FM}_b(x,i), \text{SM}_b(x,i) \; \forall \; i \in [1, \log n] \rangle$.

Note that if x is ancestor of y, and $P(x) = P(y)$, then $\text{FM}(x,y)$ and $\text{SM}(x,y)$ can be computed by just looking at the strings $\delta(x)$ and $\delta(y)$ as follows. We first extract out $depth_T(x)$ and $depth_T(y)$ respectively from $\delta(x)$ and $\delta(y)$. Now let k be the greatest integer such that $2^k \leq depth_T(y) - depth_T(x) - 1$. Then,

$$\text{FM}(x,y) = \min_1\{wt(y), \text{FM}_b(x,k), \text{FM}_a(y,k)\}, \text{ and} \tag{3.1}$$

$$\text{SM}(x,y) = \min_2\{wt(y), \text{FM}_b(x,k), \text{FM}_a(y,k), \text{SM}_b(x,k), \text{SM}_a(y,k)\} \tag{3.2}$$

For the case when x, y belongs to different paths in $\mathcal{P}$, observe that if x is ancestor of y, then $\text{PATH}_T(x,y)$ can be obtained by joining together at most $\log n$ different paths from $\mathcal{P}$.

Consider any vertex $x \in T$. Let $P_0 = P(x)$, and $P_1, \ldots, P_t$ be the paths in $\mathcal{P}$ that starts from an ancestor of x. From Lemma 3.2 it follows that t must be bounded by $\log n$. For $i = 1$ to t, let x_i and y_i be the vertices in $\text{PATH}_T(s,x) \cap P_i$ of respectively the maximum and the minimum depth value. (See Figure 3.5). We define label of x, denoted by $\ell(x)$, as the string of $O(\log^3 n)$ bits obtained by concatenating together all $\delta(x_i)$'s and $\delta(y_i)$'s, where i ranges from 0 to t.

- $\ell(x) := \langle \delta(x_i), \delta(y_i) \; \forall \; i \in [0,t] \rangle$, where x_i, y_i and t are as defined above.

Now, let z, x be a pair of query vertices such that z is ancestor of x. We give below the steps

for computing $FM(z, x)$ and $SM(z, x)$ by using labels $\ell(x)$ and $\ell(z)$.

1. Let $W = \emptyset$.
2. Compute t = number of paths in $\mathcal{P}$ by scanning the string $\ell(x)$. Let $P_0, P_1, .., P_t$ denote these paths, and x_i, y_i's be the corresponding vertices.
3. Repeat for $i = 0$ to t.
   – Add $wt(x_i)$ to set W.
   – If $depth_T(x_i) \leq depth_T(z) \leq depth_T(y_i)$, i.e. $z \in \text{PATH}_T(y_i, x_i)$, then compute $\text{FM}(z, x_i)$ and $\text{SM}(z, x_i)$ using Equations 3.1, 3.2 and add them to W. Also break the loop.
   – If $z \notin \text{PATH}_T(y_i, x_i)$, then compute $\text{FM}(y_i, x_i)$ and $\text{SM}(y_i, x_i)$ using Equations 3.1, 3.2 and add them to W.
4. Return $\text{FM}(z, x) = min_1(W)$ and $\text{SM}(z, x) = min_2(W)$.

We conclude with the following theorem.

Theorem 3.3. *Given a rooted tree T on n vertices and vertex weights in range $[0, n^2]$ we can design a labeling scheme of $O(\log^3 n)$ bits such that for any two vertices x, y, $\text{FM}(x, y)$ and $\text{SM}(x, y)$ can be computed in polylogarithmic time by just looking at the labels of x, y.*

Remark 3.2. *If in addition to weight $wt(x)$, each vertex x also has a tag field of $O(\log n)$ bits, say $\text{TAG}(x)$, then while reporting $min_1(\cdot)$ we can also report the corresponding vertex's tag. This can be easily done by defining a new weight function $wt'(x)$ which is concatenation of both $wt(x)$ and $\text{TAG}(x)$. Unless explicitly stated we assume $\text{TAG}(x)$ to be empty. The tags of two different vertices may be either same or distinct.*

3.6.2 Labeling Scheme for answering Reachability Queries

In this subsection we describe labeling scheme for answering reachability queries from s after two vertex failures. The label of a vertex x, denoted by $\sigma(x)$, will be a concatenating of many small strings $\sigma_i(x)$'s. We first define labels to check ancestor-descendant relations in trees T_1 and T_2. Consider a DFS traversal of one of the trees, say T_1. A vertex x will be an ancestor of vertex y in T_1, if and only if the start-time of x is less than the start-time of y, and the finish-time of x is greater than the finish-time of y. We thus define strings $\sigma_1(\cdot)$ to $\sigma_4(\cdot)$ as follows.

- $\sigma_1(x) :=$ start-time of x in T_1.
- $\sigma_2(x) :=$ finish-time of x in T_1.
- $\sigma_3(x) :=$ start-time of x in T_2.
- $\sigma_4(x) :=$ finish-time of x in T_2.

We also define the labels $\sigma_5(\cdot)$ to $\sigma_6(\cdot)$ to store depth fields.

- $\sigma_5(x) :=$ depth of x in T_1.
- $\sigma_6(x) :=$ depth of x in T_2.

Let u be a query vertex, and f_1, f_2 be a pair of failures. As in Section 3.5, let $u_0, u_1, ..., u_k$ be the dominators of u with $u_0 = s$ and $u_k = u$. We now consider following steps of our dual fault tolerant reachability oracle.

Step 1. *Check if condition $\mathcal{C}$ is satisfied or not.*
This can be done by just comparing the labels $\sigma_1(\cdot)$ to $\sigma_4(\cdot)$ of vertices f_1, f_2 and u. Let us assume that condition $\mathcal{C}$ holds true. Let $i, j \in [1, k]$ be such that $f_1 \in \text{PATH}_{T_1}(\bar{u}_{i-1}, \bar{u}_i)$ and $f_2 \in \text{PATH}_{T_2}(\bar{u}_{j-1}, \bar{u}_j)$.

Step 2. *Check whether $i = j$, or in other words $u_{i-1} = u_{j-1}$.*
Recall that u_{i-1} is the dominator of u lying above f_1 in T_1, and u_{j-1} is the dominator of u lying above f_2 in T_2. These can also be defined as below.

$$u_{i-1} = \text{ arg min}_y\{depth_{T_1}(y) \mid y = \text{IDOM}(x) \text{ for some } x \in \text{PATH}_{T_1}(\bar{f}_1, u)\}$$

$$u_{j-1} = \text{ arg min}_y\{depth_{T_2}(y) \mid y = \text{IDOM}(x) \text{ for some } x \in \text{PATH}_{T_2}(\bar{f}_1, u)\}$$

So $\sigma_7(\cdot)$ and $\sigma_8(\cdot)$ are defined as follows.

- $\sigma_7(x) :=$ label for reporting $\langle \text{min}_1(\cdot), \text{TAG}\rangle$ on tree-paths of T_1 when $\text{TAG}(x) = \text{IDOM}(x)$ and $wt(x) = depth_{T_1}(\text{IDOM}(x))$.
- $\sigma_8(x) :=$ label for reporting $\langle \text{min}_1(\cdot), \text{TAG}\rangle$ on tree-paths of T_2 when $\text{TAG}(x) = \text{IDOM}(x)$ and $wt(x) = depth_{T_2}(\text{IDOM}(x))$.

Now if $i \neq j$, then u will be reachable from s, thus lets assume $i = j$. Let u_i be represented by v, and so $u_{i-1} = \text{IDOM}(v)$. Also let x_1 be the first descendant of f_1 on $\text{PATH}_{T_1}(s, v)$ whose immediate dominator is equal to $\text{IDOM}(v)$. Similarly, let x_2 be the first descendant of f_2 on $\text{PATH}_{T_1}(s, v)$ whose immediate dominator is equal to $\text{IDOM}(v)$. Note that we cannot have access to labels of vertices x_1, x_2 and v. Thus, the construction will be not be a straight forward implementation of Algorithm 3.2.

Step 3. *Check if there exists a vertex $w \in S_B(v)$ for which $\text{IDOM}(w) \in S_A(v) \backslash \{\text{IDOM}(v)\}$.*
Since v is dominator of all vertices in $S_B(u) \backslash S_B(v)$, we can even search for w in the whole set $S_B(u)$. If w lies on $\text{PATH}_{T_1}(s, u)$, then its immediate-dominator will have second minimum depth among immediate-dominators of all vertices in $\text{PATH}_{T_1}(\bar{f}_1, u)$. So we just need to compute $\text{min}_2\{depth_{T_1}(\text{IDOM}(x)) \mid x \in \text{PATH}_{T_1}(\bar{f}_1, u)\}$ and see if it is less than $depth_{T_1}(f_1)$. Similar procedure is required for tree T_2. We thus define the labels $\sigma_9(\cdot)$ and $\sigma_{10}(\cdot)$ as follows.

- $\sigma_9(x) :=$ label for reporting $\text{min}_2(\cdot)$ on tree-paths of T_1 when $wt(x) = depth_{T_1}(\text{IDOM}(x))$.
- $\sigma_{10}(x) :=$ label for reporting $\text{min}_2(\cdot)$ on tree-paths of T_2 when $wt(x) = depth_{T_2}(\text{IDOM}(x))$.

Step 4. *Check if either $D^1(v)$ starts from an ancestor of f_1 in T_1, or $D^2(v)$ starts from an ancestor of f_2 in T_2.*
Note that immediate dominator of each vertex $x \in \text{PATH}_{T_1}(\bar{f}_1, u)$ is either equal to $\text{IDOM}(v)$, or a descendant of $\text{IDOM}(v)$ in T_1. Moreover, among all vertices in $\text{PATH}_{T_1}(\bar{f}_1, u)$ whose immediate dominator is same as $\text{IDOM}(v)$, vertex v has the maximum depth. Thus any TAG of v can filtered out by setting $wt(x)$ as the ordered pair $\langle depth_{T_1}(\text{IDOM}(x)), n - depth_{T_1}(x)\rangle$. Here for $i = 1, 2$, we need to compute $\beta^i(v) = depth_{T_i}(\text{first vertex of } D^i(v))$ and compare it with $depth_{T_i}(f_i)$. We thus define $\sigma_{11}(x)$ and $\sigma_{11}(x)$ respectively as follows with TAG value as $\beta^1(x)$ and $\beta^2(x)$.

- $\sigma_{11}(x) :=$ label for reporting $\langle \text{min}_1(\cdot), \text{TAG}\rangle$ on tree-paths of T_1 when $\text{TAG}(x) = \beta^1(x)$ and $wt(x) = \langle depth_{T_1}(\text{IDOM}(x)), n - depth_{T_1}(x)\rangle$.
- $\sigma_{12}(x) :=$ label for reporting $\langle \text{min}_1(\cdot), \text{TAG}\rangle$ on tree-paths of T_1 when $\text{TAG}(x) = \beta^2(x)$ and $wt(x) = \langle depth_{T_1}(\text{IDOM}(x)), n - depth_{T_1}(x)\rangle$.

Step 5. *Check if there exists a vertex* $w \in \text{PATH}_{T_1}(\bar{f}_1, u)\backslash\{x_1\}$ *such that* $\text{IDOM}(w) = \text{IDOM}(v)$ *and* $\text{AD}_1^i(w)$ *starts from an ancestor of* f_i *in* T_i*, for some* $i \in \{1, 2\}$.
Lets first consider the case $i = 1$. Then we to search for a vertex $x \in \text{PATH}_{T_1}(\bar{f}_1, u)\backslash\{x_1\}$ for which $wt(x) = \langle depth_{T_1}(\text{IDOM}(x)), \gamma_1^1(x)\rangle$ is minimum, and check if $\gamma_1^1(x)$ is less than $depth_{T_1}(f_1)$. We add $depth_{T_1}(\text{IDOM}(x))$ in the beginning of weight function so as to filter out those vertices whose immediate dominator is same as $\text{IDOM}(v)$.

The problem here is that we can perform minima queries on tree path $\text{PATH}_{T_1}(\bar{f}_1, u)$, but do not have any method to exclude out vertex x_1. So we append the field $depth_{T_1}$(pseudo-parent of x in T_1) at the end of weight function, and query for both $\min_1$ and $\min_2$. By doing this we will get two minimums, and by comparing the depth of the pseudo-parents, we can easily filter out which one is from x_1. (Recall that pseudo-parent of x_1 lies in $\text{PATH}_{T_1}(s, f_1)$). So we define the labels $\sigma_{13}(\cdot)$ and $\sigma_{14}(\cdot)$ as follows.

- $\sigma_{13}(x)$:= label for reporting $\min_1(\cdot)$ and $\min_2(\cdot)$ on tree-paths of T_1 when $wt(x) = \langle depth_{T_1}(\text{IDOM}(x)),\ \gamma_1^1(x),\ depth_{T_1}(\text{pseudo-parent of } x \text{ in } T_1)\rangle$.

- $\sigma_{14}(x)$:= label for reporting $\min_1(\cdot)$ and $\min_2(\cdot)$ on tree-paths of T_1 when $wt(x) = \langle depth_{T_1}(\text{IDOM}(x)),\ \gamma_1^2(x),\ depth_{T_1}(\text{pseudo-parent of } x \text{ in } T_1)\rangle$.

Step 6. *Check if there exists a vertex* $w \in \text{PATH}_{T_2}(\bar{f}_2, u)\backslash\{x_2\}$ *such that* $\text{IDOM}(w) = \text{IDOM}(v)$ *and* $\text{AD}_2^i(w)$ *starts from an ancestor of* f_i *in* T_i*, for some* $i \in \{1, 2\}$.
This can be taken care in exactly similar manner as in Step 5. The labels $\sigma_{15}(\cdot)$ and $\sigma_{16}(\cdot)$ are defined as follows.

- $\sigma_{15}(x)$:= label for reporting $\min_1(\cdot)$ and $\min_2(\cdot)$ on tree-paths of T_2 when $wt(x) = \langle depth_{T_2}(\text{IDOM}(x)),\ \gamma_2^1(x),\ depth_{T_2}(\text{pseudo-parent of } x \text{ in } T_2)\rangle$.

- $\sigma_{16}(x)$:= label for reporting $\min_1(\cdot)$ and $\min_2(\cdot)$ on tree-paths of T_2 when $wt(x) = \langle depth_{T_2}(\text{IDOM}(x)),\ \gamma_2^2(x),\ depth_{T_2}(\text{pseudo-parent of } x \text{ in } T_2)\rangle$.

The final label of a vertex x, denoted by $\sigma(x)$ is the concatenation of all the smaller strings $\sigma_{11}(x)$ to $\sigma_{16}(x)$ described above. We conclude with the following theorem.

Theorem 3.4. *A directed graph $G = (V, E)$ on n vertices can be preprocessed for a source vertex s to compute labels of $O(\log^3 n)$ bits such that for any two failing vertices f_1, f_2 and a destination vertex v, whether v is reachable from s in $G\backslash\{f_1, f_2\}$ can be determined in polylogarithmic time by only processing the labels associated with f_1, f_2 and v.*

Chapter 4

Fault Tolerant Subgraph for Single Source Reachability

In this chapter we address the problem of constructing a sparse subgraph that preserves the reachability from a designated source s even after k failures. The following definition characterizes this subgraph precisely.

Definition 4.1 (**k**-FTRS)**.** *Let $G = (V, E)$ be a directed graph and $s \in V$ be a designated source vertex. A subgraph H of G is said to be a k-Fault Tolerant Reachability Subgraph (k-FTRS) for G if for any subset $F \subseteq E$ of at most k edges, a vertex $v \in V$ is reachable from s in $G \setminus F$ if and only if v is reachable from s in $H \setminus F$.*

Existence of a sparse k-FTRS and its efficient construction is important from the perspective of theoretical as well as applied computer science. Moreover, reachability lies at the core of many other graph problems like strong-connectedness, dominators [LT79], double dominators [TD05], etc. Therefore obtaining a sparse k-FTRS may help in obtaining fault tolerant solution for these problems as well. The only previously known result for k-FTRS was for $k = 1$. Given a DFS tree T rooted at s, it is straightforward using the ideas of [LT79] to compute a 1-FTRS with at most $2n$ edges.

In this chapter we present an efficient algorithm for computing a sparse k-FTRS for any $k > 1$. We prove the following theorem.

Theorem 4.1. *Let G be a directed graph on n vertices, m edges with a designated source vertex s.*

*Then for any given integer $k \geq 1$, there exists an $O(2^k mn)$ time algorithm that computes a k-*FTRS *for G with at most $2^k n$ edges. Moreover, the in-degree of each vertex in this k-*FTRS *is bounded by 2^k.*

We also show that the $2^k n$ bound on the size of k-FTRS is tight by proving the following theorem.

Theorem 4.2. *For any positive integers n, k with $n \geq 2^k$, there exists a directed graph on n vertices whose k-*FTRS *must have $\Omega(2^k n)$ edges.*

The above theorems also hold in the case when the k-FTRS is defined with respect to vertex failures instead of edge failures.

Our sparse construction of k-FTRS implies solutions to the following problems in a straightforward manner.

1. ***Strong connectedness of a graph.*** We show that it is possible to preprocess G in polynomial time to build an $O(2^k n)$ size data structure that, after the failure of any set F of k edges or vertices, can determine in $O(2^k n)$ time if the strongly connected components of graph $G \setminus F$ are the same as that of graph G.

2. ***Fault tolerant dominator tree.*** We show that any given directed graph G with a source vertex s can be preprocessed in polynomial time to build an $O(2^k n)$ size data structure that, after the failure of any set F of k edges or vertices, can report the dominator tree of $G \setminus F$ in $O(2^k n)$ time.

Besides the above applications, our techniques reveal an interesting connection between two seemingly unrelated structures: farthest min-cut and k-FTRS. The *farthest* min-cut is a very basic concept in the area of flows and cuts, and was already introduced by Ford and Fulkerson [FF62].

4.1 Preliminaries

Let us start by defining some of the notations that will be used throughout this chapter.

- $E(f)$: Edges of the graph G carrying a non-zero flow for a given flow f.

- $E(P)$: Edges lying on a path P.
- $E(A)$: Edges of the graph G whose both endpoints lie in set A, where $A \subseteq V$.
- MAX-FLOW(H, S, t): The value of the maximum flow in graph H, where the source is a set of vertices S and destination is a single vertex t.

Our algorithm for computing a k-FTRS will involve the concepts of max-flow, min-cut, and edge disjoint paths. So, at times, we will visualize the same graph G as a network with unit edge capacities. The following result follows directly from Integrality of max-flows [CLRS09].

Theorem 4.3. *For any positive integer α, there is a flow from a source set S to a destination vertex t of value α if and only if there are α edge disjoint paths originating from set S and terminating at t.*

We now give definition of an (S, t)-min-cut.

Definition 4.2. *A set of edges $C \subseteq E$ is said to be an (S, t)-cut if each path from any $s \in S$ to t must pass through at least one edge from C. The size of a cut C is the number of edges present in C. An (S, t)-cut of smallest size is called (S, t)-min-cut.*

The following definition introduces the notion of FTRS from perspective of a single vertex $v \in V$.

Definition 4.3. *Given a vertex $v \in V$ and an integer $k \geq 1$, a subgraph $G' = (V, E')$, $E' \subseteq E$ is said to be k-FTRS(v) if for any set F of k edge failures, the following condition holds: v is reachable from s in $G \backslash F$ if and only if v is reachable from s in $G' \backslash F$.*

This definition allows us to define k-FTRS in an alternative way as follows.

Definition 4.4. *A subgraph H of G is a k-FTRS if and only if H is k-FTRS(v) for each v in G.*

4.2 Overview

Our starting point is a lemma (referred as Locality Lemma) that allows us to focus on a single vertex for computing k-FTRS. It essentially states that if there exists an algorithm that for any vertex v computes a k-FTRS(v) in which in-degree of v is bounded by small constant c, then

on applying this algorithm recursively on an arbitrary sequence of vertices of G, we can get a k-FTRS for G in which in-degree of all the vertices is bounded by same constant c.

Thus the problem reduces to computing a k-FTRS(t) with at most 2^k incoming edges for any $t \in V$. The construction of a k-FTRS(t) employs farthest min-cut. Recall that a (s,t)-cut C partitions the vertices into two sets: one containing the source, and the other containing the sink. The farthest min-cut is the (unique) min-cut for which the set containing the source is of largest size. We now provide the main idea underlying the construction of a k-FTRS(t).

If the max-flow from s to t in G is $k+1$ or greater, then we can define k-FTRS(t) to be any $k+1$ edge disjoint paths from s to t. In order to convey the importance of farthest min-cut, let us consider the case when max-flow is exactly k. Let us suppose that there exists a path P from s to t in $G \setminus F$, where F is the set of failing edges. Then P must pass through an edge, say (a_i, b_i), of the farthest min-cut. It follows from the properties of the farthest min-cut that if we include vertex b_i in the source then the max-flow increases (see Lemma 4.3). That is, we get at least $k+1$ edge disjoint paths from the set $\{s, b_i\}$ to t. Note that one of these paths, say Q, must be intact even after k failures. Though Q may start from b_i (instead of s), but it is not problematic as the concatenation $P[s, b_i] :: Q$ will be preserved in $G \setminus F$. This suggests that a subgraph H of G that contains $k+1$ edge disjoint paths from $\{s, b_i\}$ to t, for each i, will serve as a k-FTRS(t).

For the case when (s,t) max-flow in G is less than k, we compute a series of farthest min-cuts built on a hierarchy of nested source sets. Our construction consists of k rounds. It starts with a source set S containing the singleton vertex s. In each iteration we add to the previous source S, the endpoints b_i's of the edges corresponding to the farthest (S,t) min-cut. The size of the cuts in this hierarchy governs the in-degree of t in k-FTRS(t). In order to get a bound on the size of these cuts, we transform G into a new graph with $O(m)$ vertices and edges, so that the following assumption holds.

Assumption 1: The out-degree of all vertices in G is at most two.

It turns out that a k-FTRS(t) for the original graph can be easily obtained by a k-FTRS(t) of the transformed graph. We provide the justification for the above assumption in the next section.

In this chapter, we describe the construction of a k-FTRS with respect to edge failures only. Vertex failures can be handled by simply splitting each vertex v into an edge (v_{in}, v_{out}), where the

incoming and outgoing edges of v are respectively directed into v_{in} and directed out of v_{out}.

4.3 The Main Tools

We now describe the main tools used in obtaining our k-FTRS.

4.3.1 Locality Lemma and Proof of Assumption 1

We first formally state and prove the locality lemma.

Lemma 4.1. *Suppose there exists an algorithm $\mathcal{A}$ and an integer c_k satisfying the following condition: Given any graph G and a vertex v in G, $\mathcal{A}$ can compute a subgraph H of G such that*

(i) *H is a k-FTRS(v), and*

(ii) *in-degree of v in H is bounded by c_k.*

Then, we can compute a k-FTRS for G with at most $c_k \cdot n$ edges.

Proof. Let $\langle v_1, \ldots, v_n \rangle$ be any arbitrary sequence of the n vertices of G. We compute k-FTRS in n rounds as follows. Let $G_0 = G$ be the initial graph. In round i, we compute a graph G_i which is a k-FTRS with in-degree of vertices $v_1, ..., v_i$ bounded by c_k. This is done as follows - (i) We compute a k-FTRS(v_i), say H, for graph G_{i-1} using algorithm $\mathcal{A}$; (ii) We set G_i to be the graph obtained from G_{i-1} by restricting the incoming edges of v_i to only those present in H. It is easy to see that in-degree of vertices $v_1, ..., v_i$ in graph G_i would be bounded by c_k. We now show using induction that the graphs $G_0, G_1, .., G_n$ are all k-FTRS for G. The base case trivially holds true. In order to show that G_i ($i > 0$) is a k-FTRS for G, it suffices to show that G_i is a k-FTRS for G_{i-1}.

Consider any set F of k edge failures in G_{i-1}. Let x be any vertex reachable from s in $G_{i-1} \setminus F$ by some path, say P. We need to show the existence of a path Q from s to x in $G_i \setminus F$. If path P does not pass through v_i then we can simply set Q as P. If P passes through v_i, then we consider the segments $P[s, v_i]$ and $P[v_i, x]$. Since G_i and G_{i-1} may differ only in incoming edges of v_i, path $P[v_i, x]$ must be intact in $G_i \setminus F$. Now H is a k-FTRS(v_i) for G_{i-1}, thus there must exist a path, say Q, from s to v_i in $H \setminus F$. Note that G_i contains H. Thus $Q :: P[v_i, x]$ is a walk from s to x in $G_i \setminus F$, and after removal of all loops from it, we get a path from s to x in $G_i \setminus F$. Hence G_i is a k-FTRS for G_{i-1}. □

We now provide a justification for Assumption 1.

Lemma 4.2. *Let $G = (V, E)$ be a graph on n vertices and m edges. Then for any $t \in V$, we can compute a graph H with $O(m)$ edges and vertices, and out-degree of each vertex bounded by two such that given a k-FTRS(t) of H (say H'), we can be easily compute a k-FTRS(t) of G (say G'). Moreover, the in-degree of vertex t in G' is the same as the in-degree of t in H'.*

Proof. Given the input graph G, we construct a new graph $H = (V_0, E_0)$ from G such that out-degree of each vertex in it is bounded by two as follows.

(i) For each u in V, construct a binary tree B_u such that the number of leaves in B_u is exactly equal to the out-degree (say $d(u)$) of u in G. Let u^r be the root of this tree, and $u^\ell_1, .., u^\ell_{d(u)}$ be its leaves.

(ii) Insert the binary tree B_u in place of out-edges of u in G as follows. We delete all the out-edges, say $(u, v_1), \ldots, (u, v_{d(u)})$, of vertex u. Next we connect u to u^r, and u^ℓ_i to v_i, for each $i \leq d(u)$.

Observe that H constructed in this manner will have $O(m)$ edges and vertices. In this process, the out-degree of each vertex in H gets bounded by two, though the in-degree of the original vertices remains unchanged. Notice that an edge in G is mapped to a path in H as follows.

$$(u, v_i) \;\mapsto\; (u, u^r)::(\text{path from } u^r \text{ to } u^\ell_i \text{ in } B_u)::(u^\ell_i, v_i)$$

We now show how to compute a k-FTRS(t) in G (say G') from a k-FTRS(t) in H (say H'). For each out-neighbor v_i of a vertex u in G, we include edge (u, v_i) in G' if and only if edge (u^ℓ_i, v_i) is present in H'. Consider any set F of k failed edges in G. Define a set F_0 of failed edges in H by adding edge (u^ℓ_i, v_i) to F_0 for each $(u, v_i) \in F$. It can be inferred from the mapping defined above that there is a path from s to t in $G' \setminus F$ if and only if there is a path from s to t in $H' \setminus F_0$. Thus graph G' is a k-FTRS(t) for graph G. Notice that in-degree(t, G') is the same as in-degree(t, H'). This shows that computing a k-FTRS(t) for $t \in V$ in G is equivalent to computing a k-FTRS(t) in graph H. □

4.3.2 Farthest Min-Cut

Definition 4.5. *Let S be a source set and t be a destination vertex. Any (S,t)-min-cut C partitions the vertex set into two sets: $A(C)$ containing S, and $B(C)$ containing t. An (S,t)-min-cut C^* is said to be the* farthest min-cut *if $A(C^*) \supsetneq A(C)$ for any (S,t)-min-cut C other than C^*. We denote C^* with* $\text{FMC}(G,S,t)$.

Ford and Fullkerson [FF62] gave an algorithm for constructing the farthest (S,t)-min-cut and also established its uniqueness. For the sake of completeness we state the following result from [FF62].

Lemma 4.3. *Let G_f be the residual graph corresponding to any max-flow f_S from S to t. Let B be the set of those vertices from which there is a path to t in G_f, and $A = V \setminus B$. Then the set C of edges originating from A and terminating to B is the unique farthest (S,t)-min-cut, and is independent of the choice of the initial max-flow f_S.*

We now state an important property of the farthest min-cut. Informally, this property claims that the max-flow in G increases by one if we add to G a new edge from the set $S \times B$. (If the new edge already exists in E, then we add one more copy of it to G).

Lemma 4.4. *Let S be a source set, t be a destination vertex, C be* $\text{FMC}(G,S,t)$*, and (A,B) be the partition of V corresponding to cut C. Let $(s,w) \in (S \times B)$ be any arbitrary edge, and $G' = G + (s,w)$ be a new graph. Then,* $\text{MAX-FLOW}(G',S,t) = 1 + \text{MAX-FLOW}(G,S,t)$*, and $C' = C \cup \{(s,w)\}$ forms a (S,t)-min-cut for graph G'.*

Proof. Let f_S be a max-flow from S to t, and G_f be the corresponding residual graph. Since $w \in B$, Lemma 4.3 implies that there exists a path from w to t in G_f. This shows that there exists a path from s to t in $G_f + (s,w)$. Note that $G_f + (s,w)$ is the residual graph for G' with respect to flow f_S. Thus $\text{MAX-FLOW}(G',S,t)$ is greater than $\text{MAX-FLOW}(G,S,t)$. Since G' is obtained by adding only one extra edge to G, the value of max-flow cannot increase by more than one, hence we get that $\text{MAX-FLOW}(G',S,t)$ is equal to $1 + \text{MAX-FLOW}(G,S,t)$.

To prove the second part, note that the existence of a path P from S to t in $G' \setminus C'$ would imply the existence of a path from S to t in G not passing through cut C. Since this cannot be possible, C' must be an (S,t)-cut for graph G'. Now $|C'| = 1 + |C| = 1 + \text{MAX-FLOW}(G,S,t) =$

MAX-FLOW(G', S, t). That is, the cardinality of C' is the same as the value of max-flow from S to t. Hence, C' is an (S, t)-min-cut. □

We state two more properties of farthest-min-cut that will be used in our construction.

Lemma 4.5. *Let s and t be a pair of vertices. Let $S \subseteq V$ such that $s \in S$ and $t \notin S$. Let f_S be a max-flow from S to t, C be* FMC(S, t)*, and (A, B) be the partition of V induced by cut C. Then we can find a max-flow, say f, from s to t such that $E(f) \subseteq E(A) \cup E(f_S)$.*

Proof. Let α be equal to MAX-FLOW(G, S, t), and β be equal to MAX-FLOW(G, s, t). Let $e_1, .., e_\alpha$ be the edges lying in the cut C. Let f' be any arbitrary max-flow from s to t. Note that C is also an (s, t)-cut. Thus, without loss of generality we can assume that $C \cap E(f') = \{e_1, .., e_\beta\}$. Now let $\{(P_i :: e_i :: P'_i) : i \leq \alpha\}$ be a set of α edge disjoint paths from S to t corresponding to max-flow f_S. Since the edges of cut C are fully saturated with respect to flow f_S, each P_i will lie entirely in $G(A)$ and each P'_i will lie entirely in $G(B)$.

Let $\{(Q_i :: e_i :: Q'_i) : i \leq \beta\}$ be set of β edge disjoint paths from s to t corresponding to flow f' such that each Q_i lies entirely in $G(A)$. Note that since C is not necessarily an (s, t)-min-cut, so a path Q'_i may pass multiple times through cut C. Now $\{(Q_i :: e_i :: P'_i) : i \leq \beta\}$ forms β edge disjoint paths from s to t. This is so because Q_i's lie entirely in $G(A)$ and P'_i's lie entirely in set $G(B)$. Let f be the flow corresponding to these paths. Then f gives a max-flow from s to t such that $E(f) \subseteq E(A) \cup E(f_S)$. □

Lemma 4.6. *Let s and t be a pair of vertices. Let $S \subseteq V$ such that $s \in S$ and $t \notin S$. Let $(\hat{A}, \hat{B})$ be the partition of V induced by* FMC(s, t)*, and (A, B) be the partition of V induced by* FMC(S, t)*. Then $B \subseteq \hat{B}$.*

Proof. Consider any vertex $x \in B$. We first show that we can find a max-flow (say f_S) from S to t, and path (say P) from x to t, such that $E(f_S) \cap E(P)$ is empty. Consider the graph $G' = G + (s, x)$. From Lemma 4.4, we have that MAX-FLOW$(G', S, t) = 1 +$ MAX-FLOW$(G, S, t) = 1 + \alpha$ (say). Consider any max-flow f'_S from S to t in G'. Notice that $E(f'_S)$ must contain the edge (s, x). On removal of this edge from f'_S, it decomposes into a flow f_S (from S to t in G of capacity α), and a path P (from x and to t in G). It is easy to verify that f_S is a max-flow in G, and $E(f_S) \cap E(P) = \emptyset$.

Now from Lemma 4.5 we have that there exists a max-flow, say f, from s to t such that $E(f) \subseteq E(A) \cup E(f_S)$. Also note that path P will lie entirely in graph $G(B)$, since edges of cut (A, B) are fully saturated by flow f_S, so if path P enters set A, it will not be able to return to vertex $t \in B$. Thus $E(f) \cap E(P)$ is also empty. Thus path P lies in residual graph corresponding to max-flow f from s to t in G. So vertex x must lie in $\hat{B}$. Hence we have $B \subseteq \hat{B}$. □

4.4 A 2-FTRS for graph G

From Lemma 4.1 it follows that in order to construct a 2-FTRS for a vertex s of graph G, it is sufficient to construct for an arbitrary vertex t a subgraph H which is a 2-FTRS(t), such that the in-degree of t in H is at most 4.

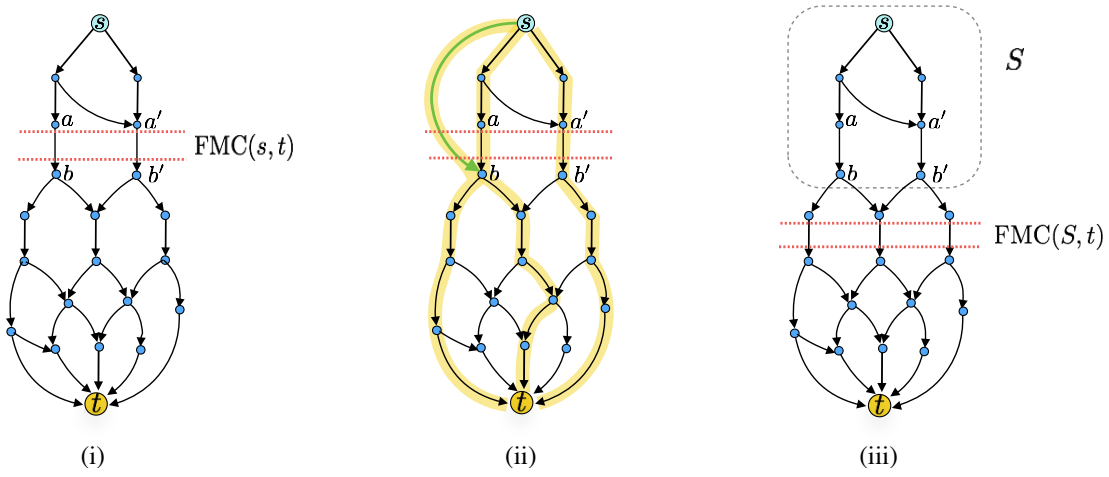


Figure 4.1: (i) Edges (a, b) and (a', b') represent the farthest min-cut from s to t. (ii) Paths highlighted in yellow color represent three edge-disjoint paths P_0, P_1, P_2 in graph $G + (s, b)$. (iii) As $b, b' \neq t$, source S is equal to $A \cup \{b, b'\}$, and FMC$(G, S, t) = 3$.

By Assumption 1 stated in Section 4.2, we have that out-degree of s is at most 2. Thus the value of max-flow from s to t must be either one or two. Below we explain how to construct a 2-FTRS(t) in each of these cases.

(i) MAX-FLOW$(G, s, t) = 2$: Let $C = \{(a, b), (a', b')\}$ be the farthest (s, t) min-cut in G. (See Figure 4.1 (i)). So after the failure of a set F of any two edges, a path from s to t (if it exists) in $G \setminus F$, must pass through C. We construct an auxiliary graph H by adding the edge (s, b) or (s, b') to G depending upon whether this path passes through (a, b) or (a', b'). Since C is the farthest

min-cut of G, it follows from Lemma 4.4 that the value of (s,t) max-flow in both the graphs $G+(s,b)$ and $G+(s,b')$ must be 3. So we denote P_0, P_1, P_2 to be any three edge disjoint paths from s to t in $G+(s,b)$ (see Figure 4.1 (ii)), and similarly P'_0, P'_1, P'_2 to be three edge disjoint paths in $G+(s,b')$. The lemma below shows how these paths can be used to compute a 2-FTRS(t).

Lemma 4.7. *Let E_t be a subset of incoming edges to t satisfying the following conditions:*

- *E_t contains last edges on paths P_0, P_1, P_2 in case $b \neq t$, and the edge (a,b) in case $b = t$.*
- *E_t contains last edges on paths P'_0, P'_1, P'_2 in case $b' \neq t$, and the edge (a',b') in case $b' = t$.*

Then the graph G^* formed by restricting the incoming edges of t in G to E_t is a 2-FTRS(t).

Proof. Consider any set F of two edge failures. Note that if t is unreachable from s in $G \setminus F$, then we have nothing to prove. So let us assume that there exists a path R from s to t in $G \setminus F$, and it passes through edge (a,b) of cut C. So, the auxiliary graph is $H = G+(s,b)$. Now if $b = t$, then R is in $G^* \setminus F$ since $(a,b) = (a,t) \in E_t$. Consider the case that $b \neq t$. Since P_0, P_1, and P_2 are edge disjoint, at least one of them is in the graph $H \setminus F$, let this path be P_0. Now either (i) P_0 is in $G \setminus F$, or (ii) the first edge on it must be (s,b). In the latter case we replace the edge (s,b) by path $R[s,b]$, so that the path $R[s,b]::P_0[b,t]$ is in $G \setminus F$. In both cases we get a path from s to t in $G \setminus F$ that enters t using only edges of the set E_t. Similar analysis can be carried out when path R passes through edge (a',b'). Hence we get that G^* is a 2-FTRS(t). □

Using the above lemma we can get a 2-FTRS(t) in which the in-degree of t is bounded by 6, by including the last edges of all six paths $P_0, P_1, P_2, P'_0, P'_1, P'_2$ in E_t. But our aim is to achieve a bound of 4. For this we construct a source set $S = A \cup (\{b,b'\} \setminus \{t\})$, where A and B forms a partition of V induced by C, and compute a max flow f_S from S to t. (See Figure 4.1 (iii)). The following lemma shows that the graph obtained by restricting the incoming edges of t to those carrying a non-zero flow with respect to f_S is a 2-FTRS(t).

Lemma 4.8. *Let $\mathcal{E}(t)$ be the set of incoming edges to t carrying a non-zero flow with respect to f_S. Then the graph $G^* = \big(G \setminus \text{IN-EDGES}(t,G)\big) + \mathcal{E}(t)$ is a* 2-FTRS(t).

Proof. In order to prove this lemma it suffices to show that there exist paths $P_0, P_1, P_2, \ldots, P'_2$ such that set $\mathcal{E}(t)$ satisfies the conditions required in Lemma 4.7. Here we show the existence of P_0, P_1, P_2. The proof for the existence of paths P'_0, P'_1, P'_2 follows in a similar manner.

Note that if $b = t$, then edge (a, b) will be a direct edge from S to t, and would thus be in f_S. In this case (a, b) is in $\mathcal{E}(t)$. So consider the case $b \neq t$. In order to compute the paths P_0, P_1, P_2 we consider the graph $G_b = G + (s, b)$. Since both the endpoints of edge (s, b) lie in S, we have that f_S is a max-flow from S to t in graph G_b, as well. Let (A_S, B_S) be the partition of V induced by any (S, t) min-cut in graph G_b. Lemma 4.5 implies that we can find a max-flow, say f, from s to t in G_b such that $E(f) \subseteq E(A_S) \cup E(f_S)$. In other words, the incoming edges to t in $E(f)$ are from the set $\mathcal{E}(t)$. Recall that Lemma 4.4 implies that value of flow f is 3. So we set P_0, P_1, P_2 to be just the three paths corresponding to flow f. □

We now show that the in-degree of t in G^* is bounded by 4. In order to prove this, it suffices to show that the value of (S, t) max-flow in G is at most 4. Now if

1. $b, b' \neq t$, then outgoing edges of b and b' will form an (S, t) cut,
2. $b = t$, then (a, b) along with outgoing edges of b' will form an (S, t) cut, and
3. $b' = t$, then (a', b') along with outgoing edges of b will form an (S, t) cut.

By Assumption 1, the out-degree of every vertex is bounded by two. Therefore, the value of (S, t) min-cut (and max-flow) can be at most 4.

(ii) MAX-FLOW$(G, s, t) = 1$: Let $C = \{(x, y)\}$ be the farthest min-cut from s to t. Then every path from s to t must pass through edge (x, y). Note that if $y = t$, then we can simply return the graph obtained by deleting all incoming edges of t except (x, t). If $y \neq t$, then the value of (y, t)-max-flow must be 2. So in this case we return a 2-FTRS(t) with y as a source (using Case i.), let this graph be G^*. It is easy to verify that G^* is a 2-FTRS(t) with s as source, and in-degree of t in G^* is bounded by 4.

This completes the construction of a 2-FTRS(t). So we have the following theorem.

Theorem 4.4. *There exists a polynomial time algorithm that for any given directed graph G on n vertices computes a* 2-FTRS *for G with at most $4n$ edges.*

4.5 Construction of a k-FTRS

In this section we prove Theorem 4.1. Let t be a vertex in G for which k-FTRS(t) from s needs to be computed. (Recall that from Lemma 4.1 it follows that this is sufficient in order to prove

Theorem 4.1.) Before we present the construction of k-FTRS(t) we state one more assumption on the graph G (in addition to Assumption 1).

Assumption 2: The out-degree of the source vertex s is 1.

The above assumption can be easily justified by adding a new vertex s' together with an edge (s', s) to G and then setting s' as the new source vertex. Algorithm 4.1 constructs a k-FTRS(t) with at most 2^k incoming edges of t.

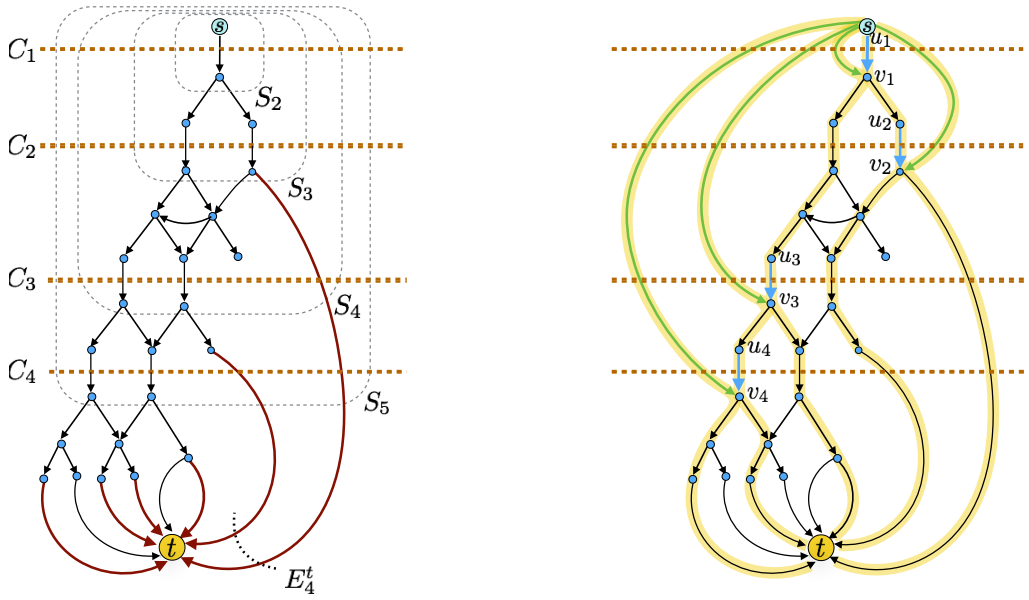
Algorithm 4.1: Algorithm for computing k-FTRS(t)

$S_1 \leftarrow \{s\}$;
for $i = 1$ *to* k **do**
  $C_i \leftarrow$ FMC(G, S_i, t);
  $(A_i, B_i) \leftarrow$ Partition(C_i);
  $S_{i+1} \leftarrow (A_i \cup \text{OUT}(A_i)) \setminus \{t\}$;
end
$f_0 \leftarrow$ max-flow from S_{k+1} to t;
$\mathcal{E}(t) \leftarrow$ Incoming edges of t present in $E(f_0)$;
Return $G^* = \big(G \setminus \text{IN-EDGES}(t, G)\big) + \mathcal{E}(t)$;

Algorithm 4.1 works as follows. It performs k iterations. In the ith iteration it computes the farthest min-cut C_i between a source set S_i and vertex t. For the 1st iteration, the source set S_1 is just vertex s. For $i \geq 1$, the source set S_{i+1} is defined by the farthest min-cut computed in the ith iteration. If (A_i, B_i) is the partition of V induced by C_i, then we set S_{i+1} as A_i union those vertices in $B_i \setminus \{t\}$ that have an incoming edge from any vertex of A_i. Notice that since $S_i \subseteq A_i$ it implies that $S_i \subseteq S_{i+1}$. After k iterations the algorithm computes a max-flow f_0 from S_{k+1} to t and sets $\mathcal{E}(t)$ to be the incoming edges of t that are in $E(f_0)$. The algorithm returns a graph G^* obtained by restricting the incoming edges of t to $\mathcal{E}(t)$. We show in the next subsection that G^* is a k-FTRS(t) with in-degree(t) bounded by 2^k. We must note that after some iteration $i < k$, the source set may become $V \setminus \{t\}$. In this case, all subsequent iterations will be redundant.

For a better understanding of the hierarchy of cuts and the source sets constructed in our algorithm, refer to Figure 4.2 (i). Note that some of the edges in a cut C_i ($i \in [1, k]$) may terminate to t, we denote this set by E_i^t. The following lemma shows that these edges are always included in our FTRS G^*.

Lemma 4.9. *For every* $1 \leq i \leq k$, $E_i^t \subseteq \mathcal{E}(t)$.



(i) (ii)

Figure 4.2: (i) The edges in brown color constitute the set $\mathcal{E}(t)$ when $k = 4$. (ii) Paths highlighted in yellow color represent 5 edge-disjoint paths in graph H.

Proof. Consider any edge $(u, t) \in E_i^t$. Since u belongs to A_i it follows from the algorithm that u is added to the source set $S_{i+1} \subseteq S_{k+1}$. Thus (u, t) is a direct edge from source S_{k+1} to t, and must be in every max-flow from S_{k+1} to t. □

4.5.1 Analysis

We now show that G^* is a k-FTRS(t). Let F be any set of k failed edges. Assume that there exists a path R from s to t in $G \setminus F$. We shall prove the existence of a path $\hat{R}$ from s to t in $G^* \setminus F$. Let $i \in [1, k]$. Since each cut C_i is an (s, t)-cut, the path R must pass through an edge, say (u_i, v_i), in C_i. Let us first consider the case when $(u_i, v_i) \in E_i^t$ for some $i \in [1, k]$. Since Lemma 4.9 implies that (u_i, v_i) is present in G^* and $v_i = t$, the path R is contained in G^*. So we can set $\hat{R}$ to R. We now turn to the case when the edge (u_i, v_i) belongs to the set $C_i \setminus E_i^t$ for every $i \in [1, k]$. In order to show the existence of a path $\hat{R}$ in G^*, we introduce a sequence of auxiliary graphs H_i's (for every $i \in [1, k + 1]$) as follows.

$$H_1 = G, \quad H_i = G + (s, v_1) + \ldots + (s, v_{i-1}), \quad i \in [2, k + 1]$$

Let $H = H_{k+1}$ be the graph obtained by adding all the edges $(s, v_1), ..., (s, v_k)$ to graph G.

(See Figure 4.2 (ii)). We will show using induction that H_i contains exactly i edge disjoint paths from s to t. Before presenting the proof, we show that for any i, the cut C_i, which is the farthest min-cut between S_i and t in G, is also the farthest min-cut between S_i and t in H_i.

Lemma 4.10. $C_i = \text{FMC}(H_i, S_i, t)$.

Proof. The cut C_i is defined as $\text{FMC}(G, S_i, t)$ and $H_i = G + \sum_{j<i}(s, v_j)$. Since the algorithm adds to the source set S_{j+1} all the vertices in $\text{OUT}(A_j) \setminus \{t\}$, the vertex $v_j \in \text{OUT}(A_j) \setminus \{t\}$ is added to $S_{j+1} \subseteq S_i$. This implies that the graph H_i is formed by adding edges such that both of their endpoints are in S_i. Hence, C_i is also equal to $\text{FMC}(H_i, S_i, t)$, and A_i, B_i form the partition of V induced by C_i in H_i. □

Lemma 4.11. $\text{MAX-FLOW}(H, s, t) = k + 1$.

Proof. We show by induction that $\text{MAX-FLOW}(H_i, s, t) = i$, for any $i \in [1, k+1]$. Note that $H_1 = G$. The base case holds since the out-degree of s is one and t is reachable from s, thus $\text{MAX-FLOW}(H_1, s, t) = 1$.

Recall that H_{i+1} is formed by adding the edge (s, v_i) to H_i, where (u_i, v_i) was an edge present in cut C_i. It follows from Lemma 4.10 that C_i is the farthest min-cut in H_i with S_i as source, and (A_i, B_i) is the partition of V induced by C_i. Let $(\hat{A}, \hat{B})$ be the partition of V induced by $\text{FMC}(H_i, s, t)$. It follows from Lemma 4.6 that $B_i \subseteq \hat{B}$. Therefore, using Lemma 4.4 we get $\text{MAX-FLOW}(H_{i+1}, s, t) = 1 + \text{MAX-FLOW}(H_i, s, t) = i + 1$. □

Next, we define H^* to be a graph obtained from H where the incoming edges of t are only those present in the set $\mathcal{E}(t)$, that is, $H^* = (H \setminus \text{IN-EDGES}(t, H)) + \mathcal{E}(t)$. The following Lemma shows that the value of max-flow remains unaffected by restricting the incoming edges of t to $\mathcal{E}(t)$.

Lemma 4.12. $\text{MAX-FLOW}(H^*, s, t) = k + 1$.

Proof. Recall that f_0 is a max-flow from S_{k+1} to t in G. Since both endpoints of the edges $(s, v_1), \ldots, (s, v_k)$ are in S_{k+1}, we get that f_0 is a max-flow from S_{k+1} to t in graph $H = G + \sum_{i=1}^{k}(s, v_i)$ as well. From Lemma 4.5 it follows that we can always find an (s, t) max-flow, say f, in H such that $E(f) \subseteq E(f_0) \cup E(A_{k+1})$. The flow f terminates at t using only edges from $\mathcal{E}(t)$, therefore, it is a flow in graph $H^* = (H \setminus \text{IN-EDGES}(t, H)) + \mathcal{E}(t)$ as well. Since f is an

(s,t) max-flow in H and max-flow cannot increase on edge removal, it follows from Lemma 4.11 that MAX-FLOW$(H^*, s, t) = k+1$. □

Note that graph H^* defined above is also equal to $G^* + \sum_{j=1}^{k}(s, v_j)$. Next, using the $k+1$ edge disjoint paths in H^* we show that G^* is a k-FTRS(t).

Lemma 4.13. *For any set F of k edges, if t is reachable from s in $G \setminus F$, then t is reachable from s in $G^* \setminus F$ as well.*

Proof. Recall that we started with assuming that R is a path from s to t in $G \setminus F$. We need to show that there exists a path $\hat{R}$ from s to t in $G^* \setminus F$. Consider the graph H^*. By Lemma 4.12 we get that there exists $k+1$ edge disjoint paths from s to t in H^*, let these be $P_0, P_1, \ldots, P_k$. Since $|F| = k$, we have that at least one of these $k+1$ paths, say P_0, must be intact in $H^* \setminus F$. Now if P_0 lies entirely in $G^* \setminus F$, then we can set $\hat{R}$ to be P_0. Thus let us assume that P_0 does not lie in $G^* \setminus F$. Since H^* is formed by adding edges $(s, v_1), .., (s, v_k)$ to G^*, we will have that the first edge on P_0 is one of the newly added edges, say (s, v_j), and the remaining path $P_0[v_j, t]$ will lie entirely in $G^* \setminus F$. Now we can simply replace edge (s, v_j) by path $R[s, v_j]$ to get path $\hat{R} = R[s, v_j]::P_0[v_j, t]$ from s to t in $G^* \setminus F$. □

We shall now establish a bound on the number of incoming edges of t in G^*.

Bounding the size of set $\mathcal{E}(t)$. Let $C_{k+1} = \text{FMC}(G, S_{k+1}, t)$. We now prove using induction that $|C_i|$ is bounded by 2^{i-1}, where $i \in [1, k]$, thus achieving a bound of 2^k on $|\mathcal{E}(t)| = |C_{k+1}|$. For the base case of $i = 1$, $|C_1| = 1$ is obvious, since the out-degree of s is one. In the following lemma we prove the induction step.

Lemma 4.14. *For any $i \geq 1$ and $i \leq k$, $|C_{i+1}| \leq 2|C_i|$.*

Proof. Let D denote the set of edges originating from S_{i+1} and terminating to $V \setminus S_{i+1}$. Since S_{i+1} contains A_i, all the edges in set E_i^t must lie in D. Now consider an edge in $(u, v) \in D \setminus E_i^t$. Note that vertex u cannot lie in A_i, because then v must be either t or lie in $(\text{OUT}(A_i) \setminus \{t\}) \subseteq S_{i+1}$. Thus edges of $D \setminus E_i^t$ must originate from vertices of the set $\text{OUT}(A_i) \setminus \{t\}$. Since $|\text{OUT}(A_i) \setminus \{t\}| \leq |C_i \setminus E_i^t|$ and the out-degree of every vertex is at most two, so we get that $|D \setminus E_i^t| \leq 2|C_i \setminus E_i^t|$. Thus, $|D| \leq |E_i^t| + 2|C_i \setminus E_i^t| \leq 2|C_i|$. Since D is an (S_i, t) cut, we get that the size of (S_i, t) min-cut must be bounded by $2|C_i|$. □

Remark 4.1. *The fact that the out-degree of each vertex is bounded above by two plays a crucial role in bounding the size of cuts C_i's in the proof of Lemma 4.14.*

Analysis of running time. We now analyze the running time of our algorithm to compute a k-FTRS(t) for any $t \in V$. The first step in computation of k-FTRS(t) is to transform G into a graph with $O(m)$ vertices and edges such that out-degree of each vertex is bounded by two. This takes $O(m)$ time (see Lemma 4.2). Next we apply Algorithm 4.1 on this transformed graph. The time complexity of Algorithm 4.1 is dominated by the time required for computing the k farthest min-cuts which is $O(\sum_{i=1}^{k} m \times |C_i|) = O(2^k m)$ (see [FF62]). Finally a k-FTRS(t) for original graph can be extracted form a k-FTRS(t) of transformed graph in $O(m)$ time (see Lemma 4.2). Thus a k-FTRS(t) for any vertex t can be computed in $O(2^k m)$ time.

Since computation of a k-FTRS requires n rounds, where in each round we compute a subgraph which is k-FTRS(v) for some $v \in V$, the total time required for constructing k-FTRS is $O(2^k mn)$. We conclude with the following theorem.

Reminder of Theorem 4.1 *Let G be a directed graph on n vertices, m edges with a designated source vertex s. Then for any given integer $k \geq 1$, there exists an $O(2^k mn)$ time algorithm that computes a k-*FTRS *for G with at most $2^k n$ edges. Moreover, the in-degree of each vertex in this k-*FTRS *is bounded by 2^k.*

4.6 Lower Bound

We shall now show that for each k, n ($n \geq 2^k$), there exists a directed graph G with $O(n)$ vertices whose k-FTRS must have $\Omega(2^k n)$ edges. Let T be a balanced binary tree of height k rooted at s. Let X be the set of leaf nodes of T, thus $|X| = 2^k$. Let Y be another set of n vertices. Then the graph G is obtained by adding an edge from each $x \in X$ to each $y \in Y$. In other words, $V(G) = V(T) \cup Y$ and $E(V) = E(T) \cup (X \times Y)$. Figure 4.3 illustrates the graph for $k = 3$.

We now show that a k-FTRS for G must contain all edges of G. It is easy to see that all edges of T must be present in a k-FTRS of G. Thus let us consider an edge $(x, y) \in X \times Y$. Let P be the tree path from s to leaf node x. Let F be the set of all those edges $(u, v) \in T$ such that $u \in P$ and v is the child of u not lying on P. Clearly $|F| = k$. Observe that x is the only leaf node of T reachable from s on the failure of the edges in set F. Thus $P::(x, y)$ is the unique path from s

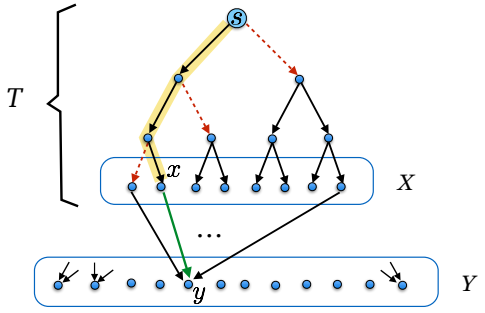


Figure 4.3: For each edge (x, y), there exist 3 edges (shown dotted) whose failure will render y unreachable from s unless (x, y) is kept in the 3-FTRS.

to y in $G \setminus F$. This shows that edge (x, y) must lie in a k-FTRS for G. This establishes a lower bound of $\Omega(2^k n)$ on the size of k-FTRS for G.

Reminder of Theorem 4.2 *For any positive integers n, k with $n \geq 2^k$, there exists a directed graph on n vertices whose k-*FTRS *must have $\Omega(2^k n)$ edges.*

4.7 Applications

In this section we present a few applications of k-FTRS.

4.7.1 Determining if a set of k edge/vertex failures alters the strong connectivity

We consider the problem of determining whether the strong connectivity relation among the vertices of G remains preserved even after the failure of any given k edges or vertices. Our construction works as follows. Let G^R denote the graph obtained by reversing each edge of graph G. Let $S_1, \ldots, S_t$ be the partition of V corresponding to the SCCs of G, and let s_i be any arbitrary vertex in S_i. For $1 \leq i \leq t$, let G_i (G_i^R) denote the graph induced by set S_i in G (G^R). For each $i \in [1, t]$, let H_i and H_i^R denote the k-FTRS for G_i and G_i^R respectively with s_i as the source vertex. Our data structure is simply the collection of the subgraphs $H_1, H_1^R, ..., H_t, H_t^R$. Given any query set F of at most k failing edges/vertices, we perform traversal from s_i in graphs $H_i \setminus F$ and $H_i^R \setminus F$, where $i \in [1, t]$. The SCC S_i is preserved if and only if the set of vertices reachable from s_i in both $H_i \setminus F$ and $H_i^R \setminus F$ is the same as the set S_i. Note that the total space used, and the query time after any k failures are both bounded by $O(2^k n)$. Thus we get the following theorem.

Theorem 4.5. *Given any directed graph $G = (V, E)$ on n vertices, a source $s \in V$, and a positive*

integer k, we can preprocess G in polynomial time to build an $O(2^k n)$ size data structure that, after the failure of any set F of k edges or vertices can determine in $O(2^k n)$ time whether the SCCs of graph $G \setminus F$ are the same as those of the graph G.

4.7.2 Fault tolerant algorithm for reporting dominator tree of a graph

Recall that a dominator tree is a tree rooted at s where each node's children are those nodes that it immediately dominates [LT79]. Buchsbaum et al. [BGK$^+$08] gave an $O(m)$ time algorithm for computing dominators and dominator tree. Here we show how this algorithm can be combined with the concept of k-FTRS to obtain a fault tolerant algorithm for reporting the dominator tree after the failure of any k edges or vertices. Let H be a $(k+1)$-FTRS for graph G. It is easy to see that on failure of any set F of k edges or vertices, the graph $H \setminus F$ is still a 1-FTRS for graph $G \setminus F$. Thus dominators of a vertex w in graph $H \setminus F$ are identical to that in graph $G \setminus F$. So in order to compute the dominator tree of $G \setminus F$ it suffices to run the algorithm of Buchsbaum et al. [BGK$^+$08] on graph $H \setminus F$. This would take time of the order of the number of edges in H, that is, $O(2^k n)$. The space used is also $O(2^k n)$ as it suffices to store just the graph H. Thus we get that the following theorem.

Theorem 4.6. *Given any directed graph $G = (V, E)$ on n vertices, a source $s \in V$, and a positive integer k, we can preprocess G in polynomial time to build an $O(2^k n)$ size data structure that, after the failure of any set F of k edges or vertices, can compute the dominator tree of $G \setminus F$ in $O(2^k n)$ time.*

Chapter 5

Fault Tolerant Oracle for Computing SCCs

Computing the strongly connected components (SCCs) of a directed graph G is one of the most fundamental problems in computer science. There are several classical algorithms for computing the SCCs in $O(m+n)$ time that are been taught in any standard undergraduate algorithms course [CLRS09]. In this chapter, we study the following natural variant of the problem in the fault tolerant setting - "What is the fastest algorithm to compute the SCCs of $G \setminus F$, where F is any set of edges or vertices?". The algorithm can use a polynomial size data structure computed in polynomial time for G during a preprocessing phase. We obtain the following result.

Theorem 5.1. *For any n-vertex directed graph G, there exists an $O(2^k n^2)$ size data structure that given any set F of at most k failing edges, can report all the SCCs of $G \setminus F$ in $O(2^k n \log^2 n)$ time. The preprocessing time required to compute this data structure is $O(2^k n^2 m)$.*

Since the time for outputting the SCCs of $G \setminus F$ is at least $\Omega(n)$, the reporting time of our data structure is optimal (up to a polylogarithmic factor) for any fixed value of k.

Georgiadis, Italiano and Parotsidis [GIP17] addressed the problem of reporting SCCs after a single edge or a single vertex failure, that is $|F| = 1$. They showed that it is possible to compute the SCCs of $G \setminus \{e\}$ for any $e \in E$ (or of $G \setminus \{v\}$ for any $v \in V$) in $O(n)$ time using a data structure of size $O(n)$ that is computed for G in a preprocessing phase in $O(m+n)$ time. Our result generalizes their result for any constant size F at the price of an extra $O(\log^2 n)$ factor in

the running time. We also use a slower preprocessing algorithm and a larger data structure.

Using the k-FTRS structure, it is relatively straightforward to obtain a data structure that, for any pair of vertices $u, v \in V$ and any set F of size k, answers in $O(2^k n)$ time queries of the form:

"Are u and v in the same SCC of $G \setminus F$?"

The data structure consists of a k-FTRS for every $v \in V$. It is easy to see that u and v are in the same SCC of $G \setminus F$ if and only if v is reachable from u in k-FTRS$(u) \setminus F$ and u is reachable from v in k-FTRS$(v) \setminus F$[1]. So the query can be answered by checking, using graph traversals, whether v is reachable from u in k-FTRS$(u) \setminus F$ and whether u is reachable from v in k-FTRS$(v) \setminus F$. The cost of these two graph traversals is $O(2^k n)$. The size of the data structure is $O(2^k n^2)$.

The challenge that we address in this chapter is given an arbitrary set F how fast *all* the SCCs of $G \setminus F$ can be computed.

5.1 Overview

We obtain our $O(2^k n \log^2 n)$-time algorithm using several new ideas. One of the main building blocks is surprisingly the following restricted variant of the problem.

Given any set F of k failed edges and any path P which is intact in $G \setminus F$, output all the SCCs of $G \setminus F$ that intersect with P (i.e. contain at least one vertex of P).

To solve this restricted version, we implicitly solve the problem of reachability from x (and to x) in $G \setminus F$, for each $x \in P$. Though it is trivial to do so in time $O(2^k n|P|)$ using k-FTRS of each vertex on P, our goal is to preform this computation in $O(2^k n \log n)$ time, that is, in running time that is independent of the length of P. For this we use a careful insight into the structure of reachability between P and V. Specifically, if $v \in V$ is reachable from $x \in P$, then v is also reachable from any predecessor of x on P, and if v is not reachable from x, then it cannot be reachable from any successor of x as well. Let w be any vertex on P, and let A be the set of vertices reachable from w in $G \setminus F$. Then we can split P at w to obtain two paths: P_1 and P_2. We already know that all vertices in P_1 have a path to A, so for P_1 we only need to focus on set

[1]Here k-FTRS(u) (resp. k-FTRS(v)) denote a k-FTRS for graph G with u (resp. v) as source.

$V \setminus A$. Also the set of vertices reachable from any vertex on P_2 must be a subset of A, so for P_2 we only need to focus on set A. This suggests a divide-and-conquer approach which along with some more insight into the structure of k-FTRS helps us to design an efficient algorithm for computing all the SCCs that intersect P.

In order to use the above result to compute all the SCCs of $G \setminus F$, we need a clever partitioning of G into a set of vertex disjoint paths. A Depth-First-Search (DFS) tree plays a crucial role here as follows. Let P be any path from root to a leaf node in a DFS tree T. If we compute the SCCs intersecting P and remove them, then the remaining SCCs must be contained in subtrees hanging from path P. So to compute the remaining SCCs we do not need to work on the entire graph. Instead, we need to work on each subtree. In order to pursue this approach efficiently, we need to select path P in such a manner that the subtrees hanging from P are of small size. The heavy path decomposition of Sleator and Tarjan [ST83] helps to achieve this objective.

Our algorithm and data structure can be extended to support insertions as well. More specifically, we can report the SCCs of a graph that is updated by insertions and deletions of k edges in the same running time.

5.2 Preliminaries

We assume that the input graph G is strongly connected, since if it is not the case, then we may apply our result to each strongly connected component of G. Below we introduce some of the notations that will be used throughout this chapter.

- T: A DFS tree of G.
- $depth(\text{PATH}_T(a, b))$: The depth of vertex a in T.
- G^R: The graph obtained by reversing all the edges in graph G.
- $\mathcal{P}$: The set of vertex disjoint paths in G obtained from a heavy path decomposition of T.

Our algorithm will require the knowledge of the vertices reachable from a vertex v as well as the vertices that can reach v. So we define a k-FTRS of both the graphs - G and G^R with respect to any source vertex v as follows.

- $\mathcal{G}(v)$: The k-FTRS of graph G with v as source obtained by Theorem 4.1.
- $\mathcal{G}^R(v)$: The k-FTRS of graph G^R with v as source obtained by Theorem 4.1.

The following lemma states that the subgraph of a k-FTRS induced by $A \subset V$ can serve as a k-FTRS for the subgraph $G(A)$ given that A satisfies certain properties.

Lemma 5.1. *Let s be any designated source and H be a k-FTRS of G with respect to s. Let A be a subset of V containing s such that every path from s to any vertex in A is contained in $G(A)$. Then $H(A)$ is a k-FTRS of $G(A)$ with respect to s.*

Proof. Let F be any set of at most k failing edges, and v be any vertex reachable from s in $G(A) \setminus F$. Since v is reachable from s in $G \setminus F$ and H is a k-FTRS of G, v must be reachable from s in $H \setminus F$ as well. Let P be any path from s to v in $H \setminus F$. Then (i) all edges of P are present in H and (ii) none of the edges of F appear on P. Since every path from s to any vertex in A is contained in $G(A)$, P must be present in $G(A)$. So every vertex of P belongs to A. This fact combined with the inferences (i) and (ii) imply that P must be present in $H(A) \setminus F$. Hence $H(A)$ is k-FTRS of $G(A)$ with respect to s. □

The next lemma is an adaptation of Lemma 10 from Tarjan's classical paper on Depth First Search [Tar72] to our needs.

Lemma 5.2. *Let T be a DFS tree of G. Let $a, b \in V$ be two vertices without any ancestor-descendant relationship in T, and assume that a is visited before b in the DFS traversal of G corresponding to tree T. Every path from a to b in G must pass through a common ancestor of a and b in T.*

Proof. Let us assume on the contrary that there exists a path P from a to b in G that does not pass through any common ancestor of a, b in T. Let z be the LCA of a, b in T, and w be the child of z lying on $\text{PATH}_T(z, a)$ in T. See Figure 5.1. Let A be the set of vertices which are either visited before w in T or lie in the subtree $T(w)$, and B be the set of vertices visited after w in T. Thus a belongs to set A, and b belongs to set B. Let x be the last vertex in P that lies in A, and y be the successor of x on P. Since none of vertices of P is a common ancestor of a and b, the edge (x, y) must belong to set $A \times B$. So the following relationship must hold true-

FINISH-TIME$(x) \leq$ FINISH-TIME$(w) <$ VISIT-TIME(y). But such a relationship is not possible since all the out-neighbors of x must be visited before the DFS traversal finishes for vertex x. Hence we get a contradiction. □

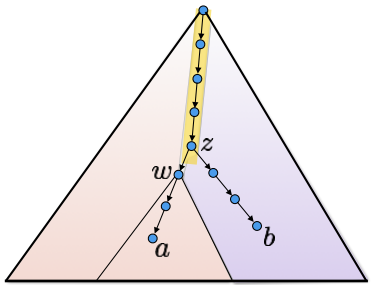


Figure 5.1: Depiction of vertices a, b, z, w and sets A (shown in orange) and B (shown in purple).

We now introduce the notion of ancestor path.

Definition 5.1. *A path* PATH$_T(a_1, b_1) \in \mathcal{P}$ *is said to be an ancestor path of* PATH$_T(a_2, b_2) \in \mathcal{P}$, *if* a_1 *is an ancestor of* a_2 *in* T.

In this chapter, we describe the algorithm for computing SCCs of graph G after any k edge failures. Vertex failures can be handled by simply splitting each vertex v into an edge (v_{in}, v_{out}), where the incoming and outgoing edges of v are directed to v_{in} and from v_{out}, respectively.

5.3 Computation of SCCs intersecting a given path

Let F be a set of at most k failing edges, and $X = (x_1, x_2, \ldots, x_t)$ be any path in G from x_1 to x_t which is intact in $G \setminus F$. In this section, we present an algorithm that outputs in $O(2^k n \log n)$ time the SCCs of $G \setminus F$ that intersect X.

For each $v \in V$, let $X^{\text{IN}}(v)$ be the vertex of X of minimum index (if exists) that is reachable from v in $G \setminus F$. Similarly, let $X^{\text{OUT}}(v)$ be the vertex of X of maximum index (if exists) that has a path to v in $G \setminus F$. (See Figure 5.2).

We start by proving certain conditions that must hold for a vertex if its SCC in $G \setminus F$ intersects X.

Lemma 5.3. *For any vertex* $w \in V$, *the SCC that contains* w *in* $G \setminus F$ *intersects* X *if and only if the following two conditions are satisfied.*

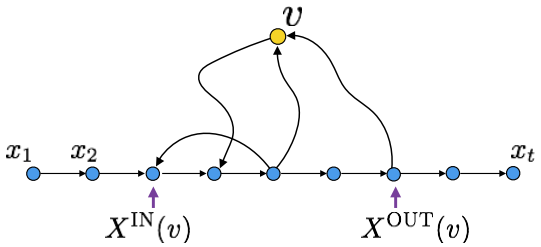


Figure 5.2: Depiction of $X^{\text{IN}}(v)$ and $X^{\text{OUT}}(v)$ for a vertex v whose SCC intersects X.

(i) Both $X^{\text{IN}}(w)$ and $X^{\text{OUT}}(w)$ are defined, and

(ii) Either $X^{\text{IN}}(w) = X^{\text{OUT}}(w)$, or $X^{\text{IN}}(w)$ appears before $X^{\text{OUT}}(w)$ on X.

Proof. Consider any vertex $w \in V$. Let S be the SCC in $G \setminus F$ that contains w and assume S intersects X. Let w_1 and w_2 be the first and last vertices of X, respectively, that are in S. Since w and w_1 are in S there is a path from w to w_1 in $G \setminus F$. Moreover, w cannot reach a vertex that precedes w_1 in X since such a vertex will be in S as well and it will contradict the definition of w_1. Therefore, $w_1 = X^{\text{IN}}(w)$. Similarly we can prove that $w_2 = X^{\text{OUT}}(w)$. Since w_1 and w_2 are defined to be the first and last vertices from S on X, respectively, it follows that either $w_1 = w_2$, or w_1 precedes w_2 on X. Hence conditions (i) and (ii) are satisfied.

Now assume that conditions (i) and (ii) are true. The definition of $X^{\text{IN}}(\cdot)$ and $X^{\text{OUT}}(\cdot)$ implies that there is a path from $X^{\text{OUT}}(w)$ to w, and a path from w to $X^{\text{IN}}(w)$. Also, condition (ii) implies that there is a path from $X^{\text{IN}}(w)$ to $X^{\text{OUT}}(w)$. Thus w, $X^{\text{IN}}(w)$, and $X^{\text{OUT}}(w)$ are in the same SCC and it intersects X. □

The following lemma states the condition under which any two vertices lie in the same SCC, given that their SCCs intersect X.

Lemma 5.4. *Let a, b be any two vertices in V whose SCCs intersect X. Then a and b lie in the same SCC if and only if $X^{\text{IN}}(a) = X^{\text{IN}}(b)$ and $X^{\text{OUT}}(a) = X^{\text{OUT}}(b)$.*

Proof. In the proof of Lemma 5.3, we show that if SCC of w intersects X, then $X^{\text{IN}}(w)$ and $X^{\text{OUT}}(w)$ are precisely the first and last vertices on X that lie in the SCC of w. Since SCCs forms a partition of V, vertices a and b will lie in the same SCC if and only if $X^{\text{IN}}(a) = X^{\text{IN}}(b)$ and $X^{\text{OUT}}(a) = X^{\text{OUT}}(b)$. □

It follows from the above two lemmas that in order to compute the SCCs in $G \setminus F$ that intersect with X, it suffices to compute $X^{\text{IN}}(\cdot)$ and $X^{\text{OUT}}(\cdot)$ for all vertices in V. It suffices to focus on

computation of $X^{\text{OUT}}(\cdot)$ for all the vertices of V, since $X^{\text{IN}}(\cdot)$ can be computed in an analogous manner by just looking at graph G^R. One trivial approach to achieve this goal is to compute the set V_i consisting of all vertices reachable from each x_i by performing a BFS or DFS traversal of graph $\mathcal{G}(x_i) \setminus F$, for $1 \leq i \leq t = |X|$. Using this straightforward approach it takes $O(2^k nt)$ time to complete the task of computing $X^{\text{OUT}}(v)$ for every $v \in V$, while our target is to do so in $O(2^k n \log n)$ time.

Observe the nested structure underlying V_i's, that is, $V_1 \supseteq V_2 \supseteq \cdots \supseteq V_t$. Consider any vertex $x_\ell, 1 < \ell < t$. The nested structure implies for every $v \in V_\ell$ that $X^{\text{OUT}}(v)$ must be on the portion $(x_\ell, \ldots, x_t)$ of X. Similarly, it implies for every $v \in V_1 \setminus V_\ell$ that $X^{\text{OUT}}(v)$ must be on the portion $(x_1, \ldots, x_{\ell-1})$ of X. This suggests a divide and conquer approach to efficiently compute $X^{\text{OUT}}(\cdot)$. We first compute the sets V_1 and V_t in $O(2^k n)$ time each. For each $v \in V \setminus V_1$, we assign NULL to $X^{\text{OUT}}(v)$ as it is not reachable from any vertex on X; and for each $v \in V_t$ we set $X^{\text{OUT}}(v)$ to x_t. For vertices in set $V_1 \setminus V_t$, $X^{\text{OUT}}(\cdot)$ is computed by calling the function Binary-Search$(1, t-1, V_1 \setminus V_t)$. See Algorithm 5.1.

Algorithm 5.1: Binary-Search(i, j, A)

```
1 if (i = j) then
2 |   foreach v ∈ A do X^OUT(v) = x_i;
3 else
4 |   mid ← ⌈(i + j)/2⌉;
5 |   B ← Reach(x_mid, A);        /* vertices in A reachable from x_mid */
6 |   Binary-Search(i, mid-1, A\B);
7 |   Binary-Search(mid, j, B);
8 end
```

In order to explain the function Binary-Search, we first state an assertion that holds true for each recursive call of the function Binary-Search. We prove this assertion in the next subsection.

Assertion 1: If Binary-Search(i, j, A) is called, then A is precisely the set of those vertices $v \in V$ whose $X^{\text{OUT}}(v)$ lies on the path $(x_i, x_{i+1}, \ldots, x_j)$.

We now explain the execution of function Binary-Search(i, j, A). If $i = j$, then we assign x_i to $X^{\text{OUT}}(v)$ for each $v \in A$ as justified by Assertion 1. Let us consider the case when $i \neq j$. In this case we first compute the index $mid = \lceil (i+j)/2 \rceil$. Next we compute the set B consisting of all the vertices in A that are reachable from x_{mid}. This set is computed using the function Reach(x_{mid}, A)

which is explained later in Subsection 5.3.2. As follows from Assertion 1, $X^{\text{OUT}}(v)$ for each vertex $v \in A$ must belong to path $(x_i, \ldots, x_j)$. Thus, $X^{\text{OUT}}(v)$ for all $v \in B$ must lie on path $(x_{mid}, \ldots, x_j)$, and $X^{\text{OUT}}(v)$ for all $v \in A \setminus B$ must lie on path $(x_i, \ldots, x_{mid\text{-}1})$. So for computing $X^{\text{OUT}}(\cdot)$ for vertices in $A \setminus B$ and B, we invoke the functions Binary-Search($i, mid\text{-}1, A \setminus B$) and Binary-Search($mid, j, B$), respectively.

5.3.1 Proof of correctness of algorithm

In this section we prove that Assertion 1 holds for each call of the Binary-Search function. Let us first see how this assertion implies that $X^{\text{OUT}}(v)$ is correctly computed for every $v \in V$. It follows from the description of the algorithm that for each $i, (1 \leq i \leq t-1)$, the function Binary-Search(i, i, A) is invoked for some $A \subseteq V$. Assertion 1 implies that A must be the set of all those vertices $v \in V$ such that $X^{\text{OUT}}(v) = x_i$. As can be seen, the algorithm in this case correctly sets $X^{\text{OUT}}(v)$ to x_i for each $v \in A$.

We now show that Assertion 1 holds true in each call of the function Binary-Search. It is easy to see that Assertion 1 holds true for the first call Binary-Search($1, t-1, V_1 \setminus V_t$). Consider any intermediate recursive call Binary-Search(i, j, A), where $i \neq j$. It suffices to show that if Assertion 1 holds true for this call, then it also holds true for the two recursive calls that it invokes. Thus let us assume A is the set of those vertices $v \in V$ whose $X^{\text{OUT}}(v)$ lies on the path $(x_i, x_{i+1}, \ldots, x_j)$. Recall that we compute index mid lying between i and j, and find the set B consisting of all those vertices in A that are reachable from x_{mid}. From the nested structure of the sets $V_i, V_{i+1}, \ldots, V_j$, it follows that $X^{\text{OUT}}(v)$ for all $v \in B$ must lie on path $(x_{mid}, \ldots, x_j)$, and $X^{\text{OUT}}(v)$ for all $v \in A \setminus B$ must lie on path $(x_i, \ldots, x_{mid\text{-}1})$. That is, B is precisely the set of those vertices whose $X^{\text{OUT}}(v)$ lies on the path $(x_{mid}, \ldots, x_j)$, and $A \setminus B$ is precisely the set of those vertices whose $X^{\text{OUT}}(v)$ lies on the path $(x_i, \ldots, x_{mid\text{-}1})$. Thus Assertion 1 holds true for the recursive calls Binary-Search($i, mid\text{-}1, A \setminus B$) and Binary-Search($mid, j, B$) as well.

5.3.2 Implementation of function Reach

The main challenge left now is to find an efficient implementation of the function Reach which has to compute the vertices of its input set A that are reachable from a given vertex $x \in X$ in $G \setminus F$. The function Reach can be easily implemented by a standard graph traversal initiated from

x in the graph $\mathcal{G}(x) \setminus F$ (recall that $\mathcal{G}(x)$ is a k-FTRS of x in G). This, however, will take $O(2^k n)$ time which is not good enough for our purpose, as the total running time of Binary-Search in this case will become $O(|X|2^k n)$. Our aim is to implement the function Reach in $O(2^k|A|)$ time. In general, for an arbitrary set A this might not be possible. This is because A might contain a vertex that is reachable from x via a single path whose vertices are not in A, therefore, the algorithm must explore edges incident to vertices that are not in A as well. However, the following lemma, that exploits Assertion 1, suggests that in our case as the call to Reach is done while running the function Binary-Search we can restrict ourselves to the set A only.

Lemma 5.5. *If* Binary-Search(i, j, A) is called and $\ell \in [i, j]$, then for each path P from x_ℓ to a vertex $z \in A$ in graph in $G \setminus F$, all the vertices of P must be in the set A.

Proof. Assertion 1 implies that A is precisely the set of those vertices in V which are reachable from x_i but not reachable from x_{j+1} in $G \setminus F$. Consider any vertex $y \in P$. Observe that y is reachable from x_i by the path $X[x_i, x_\ell]::P[x_\ell, y]$. Moreover, y is not reachable from x_{j+1}, because otherwise z will also be reachable from x_{j+1}, which is not possible since $z \in A$. Thus vertex y is in the set A. □

Lemma 5.5 and Lemma 5.1 imply that in order to find the vertices in A that are reachable from x_{mid}, it suffices to do traversal from x_{mid} in the graph G_A, the induced subgraph of A in $\mathcal{G}(x_{mid}) \setminus F$, that has $O(2^k|A|)$ edges. Therefore, based on the above discussion, Algorithm 5.2 given below, is an implementation of function Reach that takes $O(2^k|A|)$ time.

Algorithm 5.2: Reach(x_{mid}, A)

```
1 H ← G(x_mid) \ F;
2 G_A ← (A, ∅);                                   /* an empty graph */
3 foreach v ∈ A do
4 |   foreach (y, v) ∈ IN-EDGES(v, H) do
5 |   |   if y ∈ A then E(G_A) = E(G_A) ∪ (y, v);
6 |   end
7 end
8 B ← Vertices reachable from x_mid obtained by a BFS or DFS traversal of graph G_A;
9 Return B;
```

The following lemma gives the analysis of running time of Binary-Search$(1, t-1, V_1 \setminus V_t)$.

Lemma 5.6. *The total running time of* Binary-Search$(1, t-1, V_1 \setminus V_t)$ is $O(2^k n \log n)$.

Proof. The time complexity of Binary-Search$(1, t-1, V_1 \setminus V_t)$ is dominated by the total time taken by all invocations of function Reach. Let us consider the recursion tree associated with Binary-Search$(1, t-1, V_1 \setminus V_t)$. It can be seen that this tree will be of height $O(\log n)$. In each call of the Binary-Search, the input set A is partitioned into two disjoint sets. As a result, the input sets associated with all recursive calls at any level j in the recursion tree form a disjoint partition of $V_1 \setminus V_t$. Since the time taken by Reach is $O(2^k|A|)$, the total time taken by all invocations of Reach at any level j is $O(2^k|V_1 \setminus V_t|)$. As there are at most $\log n$ levels in the recursion tree, the total time taken by Binary-Search$(1, t-1, V_1 \setminus V_t)$ is $O(2^k n \log n)$. □

We conclude with the following theorem.

Theorem 5.2. *Let F be any set of at most k failed edges, and $X = \{x_1, x_2, \ldots, x_t\}$ be any path in $G \setminus F$. If we have prestored the graphs $\mathcal{G}(x)$ and $\mathcal{G}^R(x)$ for each $x \in X$, then we can compute all the SCCs of $G \setminus F$ which intersect with X in $O(2^k n \log n)$ time.*

5.4 Main Algorithm

In the previous section we showed that given any path P, we can compute all the SCCs intersecting P efficiently, if P is intact in $G \setminus F$. In the case that P contains ℓ failed edges from F then P is decomposed into $\ell + 1$ paths, and we can apply Theorem 5.2 to each of these paths separately to get the following theorem:

Theorem 5.3. *Let P be any given path in G. Then there exists an $O(2^k n|P|)$ size data structure that for any arbitrary set F of at most k edges computes the SCCs of $G \setminus F$ that intersect the path P in $O((\ell+1)2^k n \log n)$ time, where ℓ ($\ell \leq k$) is the number of edges in F that lie on P.*

Notice that in order to use Theorem 5.3 to efficiently compute SCCs of $G \setminus F$ we chose $\mathcal{P}$ to be a heavy path decomposition of DFS tree T. Choosing T as a DFS tree helps us because of the following reason: let P be any root-to-leaf path, and suppose we have already computed the SCCs in $G \setminus F$ intersecting P. Then each of the remaining SCCs must be contained in some subtree hanging from path P. The following lemma formally states this fact.

Lemma 5.7. *Let F be any set of failed edges, and $\text{PATH}_T(a, b)$ be any path in $\mathcal{P}$. Let S be any SCC in $G \setminus F$ that intersects $\text{PATH}_T(a, b)$ but does not intersect any ancestor path of $\text{PATH}_T(a, b)$*

in $\mathcal{P}$. Then all the vertices of S must lie in the subtree $T(a)$.

Proof. Consider a vertex u on $\text{PATH}_T(a, b)$ whose SCC S_u in $G \setminus F$ is not completely contained in the subtree $T(a)$. We show that S_u must contain an ancestor of a in T, thereby proving that it intersects an ancestor-path of $\text{PATH}_T(a, b)$ in $\mathcal{P}$. Let v be any vertex in S_u that is not in the subtree $T(a)$. Let $P_{u,v}$ and $P_{v,u}$ be paths from u to v and from v to u, respectively, in $G \setminus F$. From Lemma 5.2 it follows that either $P_{u,v}$ or $P_{v,u}$ must pass through a common ancestor of u and v in T. Let this ancestor be z. Notice that all the vertices of $P_{u,v}$ and $P_{v,u}$ must lie in S_u. Therefore, u and z are in the same SCC in $G \setminus F$. Moreover, since $v \notin T(a)$ and $u \in T(a)$, their common ancestor z in T is an ancestor of a. Since $z \in S_u$ and it is an ancestor of a in T, the lemma follows. □

Lemma 5.7 suggests that if we process the paths from $\mathcal{P}$ in the non-decreasing order of their depths, then in order to compute the SCCs intersecting a path $\text{PATH}_T(a, b) \in \mathcal{P}$, it suffices to focus on the subgraph induced by the vertices in $T(a)$ only. This is because the SCCs intersecting $\text{PATH}_T(a, b)$ that do not completely lie in $T(a)$ would have already been computed during the processing of some ancestor path of $\text{PATH}_T(a, b)$.

Algorithm 5.3: Compute SCC(G, F)

```
1  C ← ∅;                                              /* Collection of SCCs */
2  W ← ∅;
3  P ← A heavy path decomposition of T, where paths are sorted in the non-decreasing order
     of their depths;
4  foreach PATH_T(a, b) ∈ P do
5  |   A ← Vertices lying in the subtree T(a);
6  |   (S_1, ..., S_t) ← SCCs intersecting PATH_T(a, b) in G(A) \ F computed using D_{a,b};
7  |   foreach i ∈ [1, t] do
8  |   |   if (S_i ⊈ W) then Add S_i to collection C and set W = W ∪ S_i;
9  |   end
10 end
11 Return C;
```

We preprocess the graph G as follows. We first compute a heavy path decomposition $\mathcal{P}$ of DFS tree T. Next for each path $\text{PATH}_T(a, b) \in \mathcal{P}$, we use Theorem 5.3 to construct the data structure for path $\text{PATH}_T(a, b)$ and the subgraph of G induced by vertices in $T(a)$. We use the notation $\mathcal{D}_{a,b}$ to denote this data structure. Our algorithm for reporting SCCs in $G \setminus F$ will use the collection of these data structures associated with the paths in $\mathcal{P}$ as follows.

Let $\mathcal{C}$ denote the collection of SCCs in $G \setminus F$ initialized to $\emptyset$. We process the paths from $\mathcal{P}$ in non-decreasing order of their depths. Let $\text{PATH}_T(a, b)$ be any path in $\mathcal{P}$ and let A be the set of vertices belonging to $T(a)$. We use the data structure $\mathcal{D}_{a,b}$ to compute SCCs of $G(A) \setminus F$ intersecting $\text{PATH}_T(a, b)$. Let these be $S_1, \ldots, S_t$. Note that some of these SCCs might be a part of some bigger SCC computed earlier. We can detect it by keeping a set W of all vertices for which we have computed their SCCs. So if $S_i \subseteq W$, then we can discard S_i, else we add S_i to collection $\mathcal{C}$. Algorithm 5.3 gives the complete pseudocode of this algorithm.

Note that, in the above explanation, we only used the fact that T is a DFS tree, and $\mathcal{P}$ could have been any path decomposition of T. We now show how the fact that $\mathcal{P}$ is a heavy path decomposition is crucial for the efficiency of our algorithm. Consider any vertex $v \in T$. The number of times v is processed in Algorithm 5.3 is equal to the number of paths in $\mathcal{P}$ that start from either v or an ancestor of v, which we know from Lemma 3.2 to be bounded by $\log n$. On applying Theorem 5.3, this immediately gives that the total time taken by Algorithm 5.3 is $O(k2^k n \log^2 n)$. In the next subsection, we do a more careful analysis and show that this bound can be improved to $O(2^k n \log^2 n)$.

5.4.1 Analysis of time complexity

For any path $\text{PATH}_T(a, b) \in \mathcal{P}$ and any set F of failing edges, let $\ell(a, b)$ denote the number of edges of F that lie on $\text{PATH}_T(a, b)$. It follows from Theorem 5.3 that the time spent in processing $\text{PATH}_T(a, b)$ by Algorithm 5.3 is $O\big((\ell(a, b) + 1) \times 2^k |T(a)| \times \log n\big)$. Hence the time complexity of Algorithm 5.3 is of the order of

$$\sum_{\text{PATH}_T(a,b) \in \mathcal{P}} (\ell(a, b) + 1) \times 2^k |T(a)| \times \log n$$

In order to calculate this we define a notation $\alpha(v, \text{PATH}_T(a, b))$ as $\ell(a, b) + 1$ if $v \in T(a)$, and 0 otherwise, for each $v \in V$ and $\text{PATH}_T(a, b) \in \mathcal{P}$. So the time complexity of Algorithm 5.3 becomes

$$2^k \log n \times \Big(\sum_{\text{PATH}_T(a,b) \in \mathcal{P}} (\ell(a, b) + 1) \times |T(a)| \Big)$$

$$= 2^k \log n \times \Big(\sum_{\text{PATH}_T(a,b) \in \mathcal{P}} \sum_{v \in V} \alpha(v, \text{PATH}_T(a, b)) \Big)$$

$$= 2^k \log n \times \Big(\sum_{v \in V} \sum_{\text{PATH}_T(a,b) \in \mathcal{P}} \alpha(v, \text{PATH}_T(a,b)) \Big)$$

Observe that for any vertex v and $\text{PATH}_T(a,b) \in \mathcal{P}$, $\alpha(v, \text{PATH}_T(a,b))$ is equal to $\ell(a,b)+1$ if a is either v or an ancestor of v, otherwise it is zero. Consider any vertex $v \in V$. We now show that $\sum_{\text{PATH}_T(a,b) \in \mathcal{P}} \alpha(v, \text{PATH}_T(a,b))$ is at most $k + \log n$. Let P_v denote the set of those paths in $\mathcal{P}$ which starts from either v or an ancestor of v. Then $\sum_{\text{PATH}_T(a,b) \in \mathcal{P}} \alpha(v, \text{PATH}_T(a,b)) = \sum_{\text{PATH}_T(a,b) \in P_v} \ell(a,b)+1$. Note that $\sum_{\text{PATH}_T(a,b) \in P_v} \ell(a,b)$ is at most k. Also Lemma 3.2 implies that the number of paths in P_v is at most $\log n$. This shows that $\sum_{\text{PATH}_T(a,b) \in \mathcal{P}} \alpha(v, \text{PATH}_T(a,b))$ is at most $k + \log n$ which is $O(\log n)$, since $k \leq \log n$.

Hence the time complexity of Algorithm 5.3 becomes $O(2^k n \log^2 n)$. We thus conclude with the following theorem.

Reminder of Theorem 5.1 *For any n-vertex directed graph G, there exists an $O(2^k n^2)$ size data structure that given any set F of at most k failing edges, can report all the SCCs of $G \setminus F$ in $O(2^k n \log^2 n)$ time. The preprocessing time required to compute this data structure is $O(2^k n^2 m)$.*

5.5 Extension to handle insertion as well as deletion of edges

In this section we extend our algorithm to incorporate insertion as well as deletion of edges. That is, we describe an algorithm for reporting SCCs of a directed graph G when there are at most k edge insertions and at most k edge deletions.

Let $\mathcal{D}$ denote the $O(2^k n^2)$ size data structure, described in Section 5.4, for handling k failures. In addition to $\mathcal{D}$, we store the two k-FTRS: $\mathcal{G}(v)$ and $\mathcal{G}^R(v)$ for each vertex v in G. Thus the space used remains the same, i.e. $O(2^k n^2)$. Now let $U = (X, Y)$ be the ordered pair of k updates, with X being the set of failing edges and Y being the set of newly inserted edges. Also let $|X| \leq k$ and $|Y| \leq k$.

Our first step is to compute the collection $\mathcal{C}$, consisting of SCCs of graph $G \setminus X$. This can be easily done in $O(2^k n \log^2 n)$ time using the data structure $\mathcal{D}$. Now on addition of set Y, some of the SCCs in $\mathcal{C}$ may get merged into bigger SCCs. Let S be the subset of V consisting of endpoints of edges in Y. Note that if the SCC of a vertex gets altered on addition of Y, then its new SCC must contain at least one edge from Y, and thus also a vertex from set S. Therefore, in order to compute SCCs of $G + U$, it suffices to recompute only the SCCs of vertices lying in the set S.

Algorithm 5.4: Find-SCCs($U = (X, Y)$)

```
1 C ← SCCs of graph G \ X computed using data structure D;
2 S ← Subset of V consisting of endpoints of edges in Y;
3 H ← ⋃_{v∈S} (G(v) + G^R(v) + Y);
4 Compute SCCs of graph H \ X using any standard static algorithm;
5 foreach v ∈ S do
6 |   Merge all the smaller SCCs of C which are contained in SCC_{H\X}(v) into a single SCC;
7 end
```

Lemma 5.8. *Let H be a graph consisting of edge set Y, and the k-*FTRS *$\mathcal{G}(v)$ and $\mathcal{G}^R(v)$, for each $v \in S$. Then $SCC_{H\setminus X}(v) = SCC_{G+U}(v)$, for each $v \in S$.*

Proof. Consider a vertex $v \in S$. Since $H \setminus X \subseteq G + U$, $SCC_{H\setminus X}(v) \subseteq SCC_{G+U}(v)$. We show that $SCC_{H\setminus X}(v)$ is indeed equal to $SCC_{G+U}(v)$.

Let w be any vertex reachable from v in $G + U$, by a path, say P. Our aim is to show that w is reachable from v in $H \setminus X$ as well. Notice that we can write P as $(P_1::e_1::P_2::e_2 \cdots e_{\ell-1}::P_\ell)$, where $e_1, \ldots, e_{\ell-1}$ are edges in $Y \cap P$ and $P_1, \ldots, P_\ell$ are segments of P obtained after removal of edges of set Y. Thus $P_1, \ldots, P_\ell$ lie in $G \setminus X$. For $i = 1$ to ℓ, let a_i and b_i be respectively the first and last vertices of path P_i. Since $a_1 = v$ and $a_2, \ldots, a_\ell \in S$, the k-FTRS of all the vertices a_1 to a_ℓ is contained in H. Thus for $i = 1$ to ℓ, vertex b_i must be reachable from a_i by some path, say Q_i, in graph $H \setminus X$. Hence $Q = (Q_1::e_1::Q_2 \cdots e_{\ell-1}::Q_\ell)$ is a path from $a_1 = v$ to $b_\ell = w$ in graph $H \setminus X$.

In a similar manner we can show that if a vertex w' has a path to v in graph $G + U$, then w' will also have path to v in graph $H \setminus X$. Thus $SCC_{H\setminus X}(v)$ must be equal to $SCC_{G+U}(v)$. □

So we compute the auxiliary graph H as described in Lemma 5.8. Note that H contains only $O(k2^k n)$ edges. Next we compute the SCCs of graph $H \setminus X$ using any standard algorithm [CLRS09] that runs in time which is linear in terms of the number of edges and vertices. This algorithm will take $O(2^k n \log n)$ time, since k is at most $\log n$. Finally, for each $v \in S$, we check if the $SCC_{H\setminus X}(v)$ has broken into smaller SCCs in $\mathcal{C}$, if so, then we merge all of them into a single SCC. We can accomplish this entire task in a total $O(nk)$ time only. This completes the description of our algorithm. For the pseudocode see Algorithm 5.4.

We conclude with the following theorem.

Theorem 5.4. *For any n-vertex directed graph G, there exists an $O(2^k n^2)$ size data structure that,*

given any set U of at most k edge insertions and at most k edge deletions, can report the SCCs of graph $G + U$ in $O(2^k n \log^2 n)$ time.

Chapter 6

Approximate Single Source Fault Tolerant Shortest Path

In this chapter, we present a comprehensive study of the problem of maintaining a multiplicative $(1+\epsilon)$-approximate shortest path from a designated source s to every $v \in V \setminus \{s\}$ in the presence of a failure of an edge or a vertex. Our construction works for any arbitrary $\epsilon \in (0,1)$. We assume that our graph is a directed weighted graph with edge weights in range $[1, W]$. We study the problem from the aspect of graph theory (sparse subgraph), data structures (oracle) and distributed algorithms (labeling and routing schemes).

Sparse subgraph. We show that for any graph G it is possible to compute a subgraph H with $O(n \log^3(n) \log_{1+\epsilon}(n\,W))$ edges such that for every $v, x \in V$, distance of v from s in $H \setminus \{x\}$ is at most $(1+\epsilon)$ times the distance from s in $G \setminus \{x\}$. Moreover, the in-degree of each vertex in H is bounded by $O(\log^4(n) \log_{1+\epsilon}(n\,W))$. We show that the size of the subgraph H is optimal (up to logarithmic factors) by proving a lower bound of $\Omega(n \log_{1+\epsilon} W)$ edges.

Demetrescu, Thorup, Chowdhury and Ramachandran [DTCR08] showed that there exist graphs for which any subgraph preserving exact distances from a designated source in the presence of failure of an edge or a vertex must require $\Omega(m)$ space. Peleg and Parter [PP13] showed that even in the restricted case of unweighted graphs there exists a lower bound of $\Omega(n^{1.5})$. Therefore, it is only natural to consider an approximation.

Oracle. We show that there exists an $O(n \log_{1+\epsilon}(n\,W))$ size oracle that for any $v \in V$ reports a $(1+\epsilon)$-approximate distance of v from s upon a failure of any $x \in V$ in $O(\log\log_{1+\epsilon}(n\,W))$ time. We show that the size of the oracle is optimal (up to logarithmic factors) by proving a lower bound of $\Omega(n \log_{1+\epsilon}(\,W)/\log n)$.

Distributed algorithms. We present two distributed algorithms. We present a single source *routing scheme* in the presence of single fault with the following characteristics. The path taken by the packet on failure of any x is a $(1+\epsilon)$-approximation of the shortest path from s in the graph $G \setminus \{x\}$. Each vertex has a label of $O(\log^4(n) \log_{1+\epsilon}(n\,W))$ bits and a routing table of $O(\log^5 n \log_{1+\epsilon}(n\,W))$ bits. When the routing is started at the source the labels of the failing vertex and the destination vertex are assumed to be known.

We present also a *labeling scheme* that assigns each vertex a label of $O(\log^2(n) \log_{1+\epsilon}(n\,W))$ bits. For any two vertices $x, v \in V$ the labeling scheme outputs a $(1+\epsilon)$-approximation of the distance of v from s in $G \setminus x$ using only the labels of x and v.

6.1 Preliminaries

Below we introduce some of the notations that will be used throughout this chapter.

- T: A shortest path tree of G rooted at s.
- $wt(u,v)$: Weight of edge (u,v) in graph G.
- $wt(P)$: The weight of path P in G, i.e., if $P = (u_0, ..., u_t)$ then $wt(P) = \sum_{i=0}^{t-1} wt(u_i, u_{i+1})$.
- $\sigma(P)$: Sequence of those vertices of path P whose incoming edge in the path is a non-tree edge.
- FREQ$(w, \mathcal{C})$: The number of sequences of set $\mathcal{C}$ in which vertex w appears.
- $G \setminus x$: The graph obtained by deleting vertex x from G.
- POWERS(c): The set of all powers of c in the range $[1, n\,W]$.

Recall the definition of a detour (Definition 2.3). We now define a special class of detours, the tree-path favoring detours, that will be used in our constructions.

Definition 6.1. *A detour D from u to v is a tree-path favoring detour if for any $a, b \in D \setminus \{u, v\}$, where a precedes b in D and a is an ancestor of b in T, it holds that the segment $D[a, b]$ is a tree path.*

It is easy to see that if a detour is not a tree-path favoring detour then we can easily convert it into a tree-path favoring detour, by repeatedly replacing segment $D(a, b)$ with the tree path $\textsc{Path}_T(a, b)$. Since T is the shortest path tree, by doing this we do not increase the weight of the detour. So, henceforth we can assume that all the detours referred in this chapter are tree-path favoring detours.

For the sake of simplicity we use the following alternative weight function (usually known as the Johnson transformation) which is quite popular in the literature of shortest paths.

$$wt^*(u, v) := wt(u, v) + dist_G(s, u) - dist_G(s, v)$$

It is easy to see that for any edge (u, v), $wt^*(u, v) \geq 0$. Also if (u, v) is a tree edge then $wt^*(u, v)$ must be 0. From the next lemma it follows that for any detour D, $wt^*(D)$ must be bounded by $n\,W$.

Lemma 6.1. *Let $a, b \in T$ be such that a is an ancestor of b, and let P be any path from a to b. Then, $wt^*(P) = wt(P) - dist_G(a, b)$.*

Proof. Let $P = (a = a_0, \ldots, a_t = b)$. So

$$\begin{aligned} wt^*(P) = &\ \textstyle\sum_{i=1}^{t} wt^*(a_{i-1}, a_i) \\ = &\ \textstyle\sum_{i=1}^{t} \big(wt(a_{i-1}, a_i) + dist_G(s, a_{i-1}) - dist_G(s, a_i)\big) \\ = &\ dist_G(s, a_0) - dist_G(s, a_t) + \textstyle\sum_{i=1}^{t} wt(a_{i-1}, a_i) \\ = &\ wt(P) + dist_G(s, a) - dist_G(s, b) = \ wt(P) - dist_G(a, b) \end{aligned}$$

Thus we get $wt^*(P)$ is equal to $wt(P) - dist_G(a, b)$. □

In the next Lemma we prove an important property of certain detours which we use for constructing our subgraph. This Lemma uses the new weight function and exemplifies its significance.

Lemma 6.2. *Let $x, v \in V$ be such that x is an ancestor of v in T. There is a shortest path from s to v in $G \setminus x$ of the form* - $\textsc{Path}_T(s, a)$::D::$\textsc{Path}_T(b, v)$, where D is a detour starting from a vertex

$a \in \text{PATH}_T(s, \bar{x})$ and terminating at a vertex $b \in \text{PATH}_T(\bar{x}, v)$ for which $wt^*(D)$ is minimum.

Proof. Let P be some shortest path from s to v in $G \setminus x$. Let a be the last vertex of P that is also in $\text{PATH}_T(s, \bar{x})$, and let b be the first successor of a in P that is in $\text{PATH}_T(\bar{x}, v)$. It is easy to see that such two vertices a, b must exist. The segment $P[a, b]$ is a detour for b, as none of its internal vertices can be an ancestor of b. Thus we can set D to be $P[a, b]$. Since T is the shortest path tree we can replace the segments $P[s, a]$ and $P[b, v]$ with $\text{PATH}_T(s, a)$ and $\text{PATH}_T(b, v)$, respectively, without increasing the total weight. Thus the path $Q = \text{PATH}_T(s, a)::D::\text{PATH}_T(b, v)$ forms a shortest path from s to v in $G \setminus x$.

Finally, note that $wt(Q) = dist_G(s, v) + (wt(D) - dist_G(a, b)) = dist_G(s, v) + wt^*(D)$. For every other path Q' of the form $\text{PATH}_T(s, a')::D'::\text{PATH}_T(b', v)$, where $a' \in \text{PATH}_T(s, \bar{x})$ and $b' \in \text{PATH}_T(\bar{x}, v)$, it holds that $wt(Q') = dist_G(s, v) + wt^*(D')$. Now since $wt(Q) = wt(P)$ and P is a shortest path it follows that D must be a detour with minimum wt^* value among all detours which starts at a vertex in $\text{PATH}_T(s, \bar{x})$ and terminates at a vertex in $\text{PATH}_T(\bar{x}, v)$. □

We now state a simple lemma that will be used to obtain a randomized construction for sparse subgraph.

Lemma 6.3. *Let $\mathcal{C}$ be a collection of at most n^2 subsets of V each of size exactly $(4c \ln n)$. If we pick a subset S of V of size n/c uniformly at random, then probability that there exists a $W \in \mathcal{C}$ for which $W \cap S$ is empty is at most $1/n^2$.*

Proof. Let $t = (4c \ln n)$. Note that there are total $\binom{n}{n/c}$ possibilities for set S. Now for any subset $A \in \mathcal{C}$, probability that $A \cap S$ is empty is

$$\frac{\binom{n-t}{n/c}}{\binom{n}{n/c}} \leq \left(\frac{n-t}{n}\right)^{n/c} = \left(1 - \frac{1}{n/t}\right)^{n/t \cdot 4 \ln n} \leq \frac{1}{n^4}$$

On applying union bound we get that probability there exists a $W \in \mathcal{C}$ for which $W \cap S$ is empty is at most $1/n^2$. □

6.2 Main Ideas

For any $\alpha > 0$, let $\text{HD}_\alpha(v)$ be a detour that originates from the highest possible ancestor of v in T and terminates at v such that its weight $wt^*(\text{HD}_\alpha(v)) \leq \alpha$, that is, there is no other detour that

ends at v and starts at a higher ancestor of v whose weight is bounded by α. We denote the first vertex of $\text{HD}_\alpha(v)$ with $\text{FIRST}_\alpha(v)$. Consider the following subgraph:

$$H = T \bigcup \Big(\bigcup_{\substack{b \in V \\ \alpha \in \text{POWERS}(1+\epsilon)}} \text{HD}_\alpha(b) \Big)$$

For any $x, v \in V$, the graph $H \setminus x$ contains a $(1+\epsilon)$-approximation of the shortest path from s to v in $G \setminus x$. This is because if the shortest path takes a detour D from a to b to avoid x, then H will contain a detour D' that starts at a or one of its ancestors, ends at b and $wt^*(D') \leq (1+\epsilon)wt^*(D)$. Based on this key observation, we are able to compute an oracle of $O(n \log_{1+\epsilon}(nW))$ size that can report a $(1+\epsilon)$-approximation of distance of any vertex from the source after the occurrence of failure in $O(\log\log_{1+\epsilon}(nW))$ time. Though the subgraph H can be seen as a fault tolerant $(1+\epsilon)$-shortest path subgraph, its size can be as large as $\Theta(m)$. This is because even a single detour may contain n edges. So storing $\text{HD}_\alpha(v)$ for each v and each α may require $\Omega(m)$ space in the worst case. In order to achieve sparseness for H, our starting point is the sub-structure property of $\text{HD}_\alpha(v)$ stated in the following lemma.

Lemma 6.4. *Let $D = \text{HD}_\alpha(v)$ be a detour for v, for some $\alpha > 0$. Then for any vertex $w \in \sigma(D)$, the segment $D[\cdot, w]$ is also a detour.*

Proof. Let u be the first vertex on D, that is, $u = \text{FIRST}_\alpha(v)$. We first show that all the vertices of D must lie in the subtree $T(u)$. Assume this is not the case, and let y be the last vertex of D that is not in the subtree $T(u)$. Also let z be the Lowest Common Ancestor (LCA) of y and u. Consider the path $D' = \text{PATH}_T(z, y)::D[y, v]$. It is easy to see that D' is a detour for v. Also the set of non-tree edges of D' is a subset of the non-tree edges of D, thus $wt^*(D') \leq wt^*(D) \leq \alpha$. But this contradicts the definition of D, as D' is a detour for v starting from an ancestor of u with wt^* at most α.

From the above discussion it follows that u must be an ancestor of w. Therefore, it suffices to show now that none of the internal vertices of $D[u, w]$ are ancestors of w. Let us suppose there exists a vertex $z \in D[u, w]$ ($z \neq u, w$) such that z is an ancestor of w. But in such a case we can replace the segment $D[z, w]$ of detour D with $\text{PATH}_T(z, w)$, thereby contradicting the fact that D is a tree path favoring detour. Hence $D[u, w]$ cannot pass through any ancestor of w, except for the starting vertex u. □

It follows from Lemma 6.4 that for any $w \in \sigma(\text{HD}_\alpha(v))$, if H contains a $(1+\epsilon)$-approximation of the detour $\text{HD}_\alpha(v)[\cdot, w]$, then on including just the suffix $\text{HD}_\alpha(v)[w, v]$, we can see that H will contain a $(1+\epsilon)$-approximation of $\text{HD}_\alpha(v)$. A simple way to include a $(1+\epsilon)$-approximation of the detour $\text{HD}_\alpha(v)[\cdot, w]$ would be to add to H the detours $\text{HD}_\beta(w)$, for each $\beta \in \text{POWERS}(1+\epsilon)$. However, to get a sparse subgraph instead of including each $\text{HD}_\beta(w)$, we can use the same trick recursively and include a further $(1+\epsilon)$-approximation of $\text{HD}_\beta(w)$. This motivates for a hierarchical computation of H in some $k(\geq 2)$ rounds where in a round we only add a short suffix of $\text{HD}_\alpha(v)$ to the subgraph H, and we move its prefix (which is a detour in itself) to the next round to be processed recursively. This constitutes the main idea for constructing a sparse subgraph and a compact routing scheme for approximate shortest paths from s under failure of any vertex.

In this chapter, we describe all our constructions with respect to vertex failure only. Edge failure can be handled by inserting a vertex, say z_{uv}, in middle of each tree edge (u, v). So the deletion of tree edge (u, v) is equivalent to deletion of vertex z_{uv}.

6.3 Sparse Subgraph

For the sake of better exposition of the algorithm, we first present the construction of a subgraph with $\widetilde{O}(n^{1.5} \log_{1+\epsilon}(n\,W))$ edges using a hierarchy of two levels. In the next subsection, we extend it to a k-level hierarchical construction that achieves a bound of $\widetilde{O}(n \log_{1+\epsilon}(n\,W))$ on the size of the subgraph.

6.3.1 Sparse subgraph with $\widetilde{O}(n^{1.5} \log_{1+\epsilon}(n\,W))$ edges

In this subsection, we give a construction of a sparse subgraph H with $\widetilde{O}(n^{1.5} \log_{1+\epsilon}(n\,W))$ edges that with high probability preserves the approximate shortest paths from s after a failure of any vertex. The underlying idea used in the construction of this subgraph is the following. We pick a small set S of vertices uniformly and at random. For each $v \in S$ and for each $\alpha \in \text{POWERS}(1+\epsilon)$, we include $\text{HD}_\alpha(v)$ in the subgraph H. We cannot afford to include $\text{HD}_\alpha(u)$ for every $u \in V \backslash S$ so we include only a *small* suffix of $\text{HD}_\alpha(u)$. Due to the random sampling used to construct S, it turns out that if $\text{HD}_\alpha(u)$ is long, then its small suffix will have a vertex, say w, from S. The detour of w concatenated with the small suffix of $\text{HD}_\alpha(u)$ will preserve $\text{HD}_\alpha(u)$ approximately. In

Algorithm 6.1 we present the pseudocode for computing a subgraph H with $\widetilde{O}(n^{1.5}\log_{1+\epsilon}(n\,W))$ edges that is based on this idea.

Algorithm 6.1: Computation of subgraph with $\widetilde{O}(n^{1.5}\log_{1+\epsilon}(n\,W))$ edges

```
1 H ← T;
2 foreach v ∈ V and α ∈ POWERS(1 + ε) do
3 |   Add the last √n non-tree edges of HD_α(v) to H;
4 end
5 S ← A uniformly random set of 4√n log_e n vertices;
6 foreach v ∈ S and α ∈ POWERS(1 + ε) do
7 |   Add all non-tree edges of HD_α(v) to H;
8 end
```

We first show that with high probability any long detour ($> \sqrt{n}$) has in its $\sqrt{n}$ length suffix at least one vertex from set S.

Lemma 6.5. *With high probability the following holds - For each $\alpha \in$ POWERS$(1+\epsilon)$ and each $v \in V$, if $\text{HD}_\alpha(v)$ contains more than $\sqrt{n}$ non-tree edges, then the last $\sqrt{n}$ vertices of $\sigma(\text{HD}_\alpha(v))$ contain a vertex from set S.*

Proof. Consider the collection $\mathcal{C}$ defined below.

$$\mathcal{C} = \{\text{last } \sqrt{n} \text{ vertices of } \sigma(\text{HD}_\alpha(v)) \mid v \in V, \alpha \in \text{POWERS}(1+\epsilon), |\sigma(\text{HD}_\alpha(v))| > \sqrt{n}\}$$

The collection $\mathcal{C}$ will contain at most n^2 sets (each of size $\sqrt{n}$), since for any vertex v we can have at most n detours corresponding to n ancestors of v in T. So, the result follows by simply applying Lemma 6.3. □

We will now show that upon a failure of any vertex x, the shortest paths from s in graph $G\backslash x$ are stretched by a factor of at most $(1+\epsilon)^2$ in $H\backslash x$.

Lemma 6.6. *For every two vertices $x, v \in V$, w.h.p,*

$$dist_{H\backslash x}(s,v) \leq (1+\epsilon)^2 dist_{G\backslash x}(s,v).$$

Proof. Let P be the shortest path from s to v in $G \setminus x$. If x is not an ancestor of v in T then P will be just the tree path from s to v in T. Thus let us consider the case when x is an ancestor of v. By Lemma 6.2, the path P can be represented as $\text{PATH}_T(s,a)::D::\text{PATH}_T(b,v)$, where D is a detour

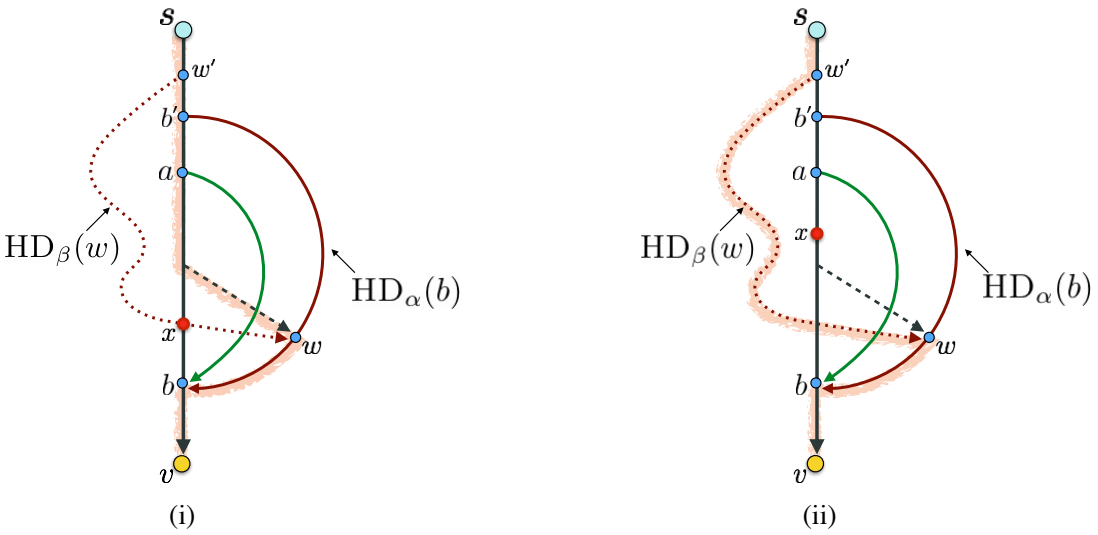


Figure 6.1: Approximate shortest path from s to v in $H \setminus x$ in the subcases: (i) x is not an ancestor of w in T, and (ii) x is an ancestor of w.

avoiding x. We take α to be the smallest power of $(1+\epsilon)$ greater than $wt^*(D)$. Let us consider the following two cases separately.

Case 1 : $\text{HD}_\alpha(b)$ contains at most $\sqrt{n}$ non-tree edges.

In this case $\text{HD}_\alpha(b)$ will lie in subgraph H. Since $\alpha \geq wt^*(D)$, $\text{FIRST}_\alpha(b)$ must be either equal to a or an ancestor of a. So instead of detour D, we can just follow the detour $\text{HD}_\alpha(b)$. Thus the concatenation $Q = \text{PATH}_T(s, \text{FIRST}_\alpha(b))::\text{HD}_\alpha(b)::\text{PATH}_T(b, v)$ forms a path from s to v in $H \setminus x$. Also, $wt^*(Q)$ is at most $(1+\epsilon)wt^*(P)$, as under the weight function wt^*, tree edges get zero weight.

Case 2 : $\text{HD}_\alpha(b)$ contains more than $\sqrt{n}$ non-tree edges.

In this case Lemma 6.5 implies that w.h.p. the last $\sqrt{n}$ vertices of $\sigma(\text{HD}_\alpha(b))$ must contain a vertex, say w, from set S. So the segment $\text{HD}_\alpha(b)[w, b]$ will lie in H because the last $\sqrt{n}$ non-tree edges of $\text{HD}_\alpha(b)$ are included in H. Also, by Lemma 6.4, the prefix $\text{HD}_\alpha(b)[\cdot, w]$ is a detour for vertex w. We further consider the following subcases. (See Figure 6.1).

(i) If x is not an ancestor of w, then simply take Q as $\text{PATH}_T(s, w)::\text{HD}_\alpha(b)[w, b]::\text{PATH}_T(b, v)$. Since $wt^*(\text{HD}_\alpha(b)[w, b]) \leq wt^*(\text{HD}_\alpha(b)) \leq (1+\epsilon)wt^*(D)$, we get that $wt^*(Q) \leq (1+\epsilon) \times wt^*(P)$.

(ii) We now consider the subcase when x is an ancestor of w in T. Notice that prefix $\text{HD}_\alpha(b)[\cdot, w]$ is not included in H, but we can take a further $(1+\epsilon)$-approximation of it. Let β be the smallest power of $(1+\epsilon)$ greater than $wt^*(\text{HD}_\alpha(b)[\cdot, w])$. Since $w \in S$, $\text{HD}_\beta(w)$ will lie in subgraph H. So we take Q as $\text{PATH}_T(s, \text{FIRST}_\beta(w))::\text{HD}_\beta(w)::\text{HD}_\alpha(b)[w, b]::\text{PATH}_T(b, v)$. See Figure 6.1(ii). Notice that $\text{HD}_\beta(w)$ cannot contain x, since $\text{FIRST}_\beta(w)$ is either same as or an ancestor of the first vertex of $\text{HD}_\alpha(b)[\cdot, w]$. Finally $wt^*(\text{HD}_\beta(w)::\text{HD}_\alpha(b)[w, b]) \leq (1+\epsilon)wt^*(\text{HD}_\alpha(b)) \leq (1+\epsilon)^2 \times wt^*(D)$. Hence, $wt^*(Q)$ is at most $(1+\epsilon)^2 \times wt^*(P)$.

In all the above cases/subcases, we were able to show that w.h.p. there exists a path Q such that $wt^*(Q) \leq (1+\epsilon)^2 \times wt^*(P)$. Now as s is ancestor of v, on adding $dist_G(s, v)$ on both sides and applying Lemma 6.1, we get that $wt(Q) \leq (1+\epsilon)^2 \times wt(P)$. □

We conclude with the following theorem.

Theorem 6.1. *Let G be a directed weighted graph on n vertices with maximum edge weight W and s be the designated source vertex. Then we can compute in polynomial time a subgraph H with $O\big(n^{1.5}\log(n)\log_{1+\epsilon}(nW)\big)$ edges such that with high probability following relation holds:*

$$\textit{For any } x, v \in V,\ dist_{H\setminus x}(s, v) \leq (1+\epsilon)^2 dist_{G\setminus x}(s, v).$$

6.3.2 Sparse subgraph with $\widetilde{O}(n\log_{1+\epsilon}(nW))$ edges

In the $\widetilde{O}(n^{1.5}\log_{1+\epsilon}(nW))$ size subgraph described in the previous subsection, we constructed a 2-level hierarchy of vertices, namely, S and V. The detour of vertices in set S and the short suffixes of detours of vertices in V could preserve every detour D up to a stretch of $(1+\epsilon)^2$. In order to further improve the size of the subgraph, we form a finer hierarchy of subsets of vertices: $S_1, \ldots, S_k$ for some $k > 2$. As we go up in this hierarchy, the size of these sets decreases and thus we can afford to store *longer* suffixes of detours from their vertices. In particular, for a given $i \in [1, k]$, S_i will have at most $n^{1-(i-1)/k}$ vertices and from each vertex $v \in S_i$, we store suffixes of $\widetilde{O}(n^{i/k}\log_{1+\epsilon}(nW))$ length. Similar to the 2-level case, a combination of k types of these detour suffixes will preserve every detour D up to a factor $(1+\epsilon)^k$. We now formalize this key idea by defining a $(1+\epsilon, t)$-preserver of a detour as follows.

Definition 6.2. *Let D be a detour from u to v, $\epsilon \in (0,1)$ be some real number, and t be some positive integer. Also let α be the smallest power of $(1+\epsilon)$ greater than or equal to $wt^*(D)$. Then a $(1+\epsilon,t)$-preserver of D is a path in G obtained as follows. (See Figure 6.2).*

1. *If $t = 1$, then a $(1+\epsilon,1)$-preserver of D is just $\text{HD}_\alpha(v)$.*
2. *If $t > 1$, then it is obtained by concatenating (i) a $(1+\epsilon,t-1)$-preserver of prefix $\text{HD}_\alpha(v)[\cdot,w]$, and (ii) the suffix $\text{HD}_\alpha(v)[w,v]$, for some vertex $w \in \sigma(\text{HD}_\alpha(v))$.*

Remark 6.1. *For above definition to be well defined we require that $\text{HD}_\alpha(v)[\cdot,w]$ is a detour for vertex w. This is indeed true since $w \in \sigma(\text{HD}_\alpha(v))$, and the proof follows from Lemma 6.4.*

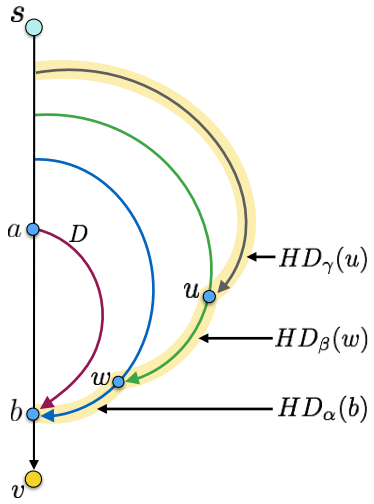


Figure 6.2: The path highlighted in yellow color depicts a $(1+\epsilon,3)$-preserver of detour D. Here α,β,γ are respectively smallest powers of $(1+\epsilon) \geqslant$ weight of (i) D, (ii) $\text{HD}_\alpha(b)[\cdot,w]$, and (iii) $\text{HD}_\beta(w)[\cdot,u]$. Note that in reality $\text{HD}_\beta(w)$ and $\text{HD}_\gamma(u)$ may not be disjoint with $\text{PATH}_T(s,b)$, however, for sake of better understanding we made them disjoint.

The following lemma shows the significance of a $(1+\epsilon,t)$-preserver.

Lemma 6.7. *Let D be a detour from a to b, and H be a subgraph of G containing tree T and a $(1+\epsilon,t)$-preserver for D. Then for any internal vertex x lying on $\text{PATH}_T(a,b)$, the graph $H \setminus x$ contains a path, say Q, from s to b such that $wt^*(Q)$ is at most $(1+\epsilon)^t \times wt^*(\text{PATH}_T(s,u)::D)$.*

Proof. We prove the lemma by applying induction on integer t. Let us denote by P the path $\text{PATH}_T(s,u)::D$, then $wt^*(P) = wt^*(D)$. Now let α be the smallest power of $(1+\epsilon)$ greater than or equal to $wt^*(D)$. Since $\alpha \geq wt^*(D)$, the first vertex on $\text{HD}_\alpha(b)$, i.e., $\text{FIRST}_\alpha(b)$ must be either

a or an ancestor of a. We first prove that the base case holds true, and later using induction give the proof for the generic case.

Base case : If $t = 1$, then $\text{HD}_\alpha(b)$ will lie in H. So path $Q = \text{PATH}_T(s, \text{FIRST}_\alpha(b))::\text{HD}_\alpha(b)$ forms a path from s to v in $H \setminus x$. Also $wt^*(Q)$ is at most $(1+\epsilon) \times wt^*(P)$.

Generic case: If $t > 1$, then there will exist a vertex $w \in \sigma(\text{HD}_\alpha(b))$ for which H contains both the suffix $\text{HD}_\alpha(b)[w, b]$ and a $(1+\epsilon, t-1)$-preserver of the prefix $\text{HD}_\alpha(b)[\cdot, w]$. We have the following two subcases depending upon whether or not x is an ancestor of w.

(i) Let us first consider the scenario when x is an ancestor of w. Though the prefix $\text{HD}_\alpha(b)[\cdot, w]$ does not lie in H, it can be seen that H contains a $(1+\epsilon, t-1)$-preserver of it. Since x is an internal vertex of $\text{PATH}_T(\text{FIRST}_\alpha(b), w)$, by induction hypothesis, $H \setminus x$ must contain a path Q_0 such that

$$wt^*(Q_0) \leq (1+\epsilon)^{t-1} wt^*\Big(\text{PATH}_T(s, \text{FIRST}_\alpha(b)) :: \text{HD}_\alpha(b)[\cdot, w]\Big)$$

Thus we choose Q to be the path $Q_0::\text{HD}_\alpha(b)[w, b]$. It is easy to see that Q is a path from s to b in H avoiding x. Also,

$$\begin{aligned} wt^*(Q) &= wt^*(Q_0) + wt^*(\text{HD}_\alpha(b)[w, b]) \\ &\leq (1+\epsilon)^{t-1} wt^*\Big(\text{PATH}_T(s, \text{FIRST}_\alpha(b)) :: \text{HD}_\alpha(b)[\cdot, w]\Big) + wt^*(\text{HD}_\alpha(b)[w, b]) \\ &\leq (1+\epsilon)^{t-1} wt^*\Big(\text{PATH}_T(s, \text{FIRST}_\alpha(b)) :: \text{HD}_\alpha(b)\Big) \\ &\leq (1+\epsilon)^{t} wt^*(\text{PATH}_T(s, u) :: D) \end{aligned}$$

Thus in this subcase we are able to show that $wt^*(Q)$ is at most $(1+\epsilon)^t wt^*(P)$.

(ii) If x is not an ancestor of w in T, then we can simply take Q to be $\text{PATH}_T(s, w)::\text{HD}_\alpha(b)[w, b]$. Notice that $wt^*(\text{HD}_\alpha(b)[w, b]) \leq wt^*(\text{HD}_\alpha(b))$ which is at most$(1+\epsilon)wt^*(D)$. Thus in this subcase $wt^*(Q)$ is at most $(1+\epsilon) \times wt^*(P) \leq (1+\epsilon)^t \times wt^*(P)$. This completes our proof. □

Lemma 6.1 together with Lemma 6.7 implies the following corollary.

Corollary 6.1. *Let D be a detour from a to b, and H be a subgraph of G containing tree T and a $(1+\epsilon, t)$-preserver for D. Then for any internal vertex x lying on $\text{PATH}_T(a, b)$, the graph*

$H \setminus x$ contains a path, say Q, from s to b whose weight (i.e. $wt(Q)$) is at most $(1+\epsilon)^t$ times $wt(\text{PATH}_T(s,u)::D)$.

We now describe the algorithm for computing the sparse subgraph H. The algorithm consists of k rounds that operate on k sets, namely, $S_1, S_2, \ldots, S_k$ as follows. Let $S_1 = V$ be the initial set. In any round $i < k$, we first compute a uniformly random subset of V of size $O(n/n^{i/k})$, and move these vertices to S_{i+1}. Next for each each $v \in S_i$ and each $\alpha \in \text{POWERS}(1+\epsilon)$,

1. If $\sigma(\text{HD}_\alpha(v))$ does not contain any vertex from S_{i+1}, then the complete detour $\text{HD}_\alpha(v)$ is added to H.

2. Otherwise, if z is the last vertex of $\sigma(\text{HD}_\alpha(v))$ that lies in S_{i+1}, then the suffix $\text{HD}_\alpha(w)[z,w]$ is added to H.

Finally in round k, for each each $v \in S_k$ and each $\alpha \in \text{POWERS}(1+\epsilon)$, we add $\text{HD}_\alpha(v)$ to H. Also the shortest path tree T is added to the graph H.

Algorithm 6.2 presents the pseudocode for this construction. In the algorithm, we use the notation $\text{SUFFIX}(\sigma, A)$ to denote the maximal suffix of σ that does not contain an element of A, where σ is any sequence of vertices and A is a subset of V. Notice that, instead of step 1 and step 2 stated above, we can equivalently just say that for each $w \in \text{SUFFIX}\big(\sigma(\text{HD}_\alpha(v)), S_{i+1}\big)$, the incoming edge of w in $\text{HD}_\alpha(v)$ is added to H.

Algorithm 6.2: Computation of subgraph with $\widetilde{O}(n \log_{1+\epsilon}(nW))$ edges

```
1  H ← T;
2  S_1 ← V(G);
3  for i = 1 to k-1 do
4  |   S_{i+1} ← A uniformly random subset of V of size O(n/n^{i/k});
5  |   foreach v ∈ S_i and α ∈ POWERS(1+ε) do
6  |   |   foreach w ∈ SUFFIX(σ(HD_α(v)), S_{i+1}) do
7  |   |   |   Add incoming edge of w in HD_α(v) to H;
8  |   |   end
9  |   end
10 end
11 foreach v ∈ S_k and α ∈ POWERS(1+ε) do Add HD_α(v) to H;
```

The following lemma shows that the subgraph H contains a $(1+\epsilon, k)$-preserver for each possible detour D in G.

Lemma 6.8. *The graph H computed by Algorithm 6.2 contains a $(1+\epsilon, k)$-preserver for each possible detour D in G.*

Proof. In order to prove the lemma it suffices to show that the following claim holds.

Claim: Let i be any index in $[1, k]$. Then for any $v \in S_{k-i+1}$, the graph H contains a $(1+\epsilon, i)$ preserver of each detour terminating at v.

We prove the above claim by applying induction on index i. To prove the base case, i.e. $i = 1$, consider any vertex $v \in S_k$. Since H contains $\text{HD}_\alpha(v)$ for each $\alpha \in \text{POWERS}(1+\epsilon)$, it is easy to see that H contains a $(1+\epsilon, 1)$ preserver of each detour terminating at v.

Now consider any index $i \in [2, k]$. Let us assume that the claim holds true for all indices $j < i$. Let v be a vertex in S_{k-i+1} and D be a detour terminating at v. We need to show that H contains a $(1+\epsilon, i)$ preserver of D. Let α be the smallest power of $(1+\epsilon)$ greater than or equal to $wt^*(D)$. Consider the detour $\text{HD}_\alpha(v)$. If $\sigma(\text{HD}_\alpha(v)) \cap S_{i+1} = \emptyset$, then H will contain the complete detour $\text{HD}_\alpha(v)$ which is a $(1+\epsilon, 1)$ preserver of D. And, we know that a $(1+\epsilon, 1)$-preserver of D is also a $(1+\epsilon, i)$-preserver of D. If $\sigma(\text{HD}_\alpha(v)) \cap S_{i+1} \neq \emptyset$, then H will contain the suffix $\text{HD}_\alpha(v)[z, v]$, where z is the last vertex in $\sigma(\text{HD}_\alpha(v))$ that lies in set S_{i+1}. Also by induction hypothesis, H will contain a $(1+\epsilon, i-1)$ preserver of prefix $\text{HD}_\alpha(v)[\cdot, z]$, say P. Now by Definition 6.2 it follows that $P{::}\text{HD}_\alpha(v)[z, v]$ is a $(1+\epsilon, i)$ preserver of detour D, which is contained in H. This shows that the claim holds for index i as well. □

Corollary 6.1 along with Lemma 6.8 implies that upon failure of any vertex x, the shortest paths from s in graph $G \backslash \{x\}$ are stretched by a factor of at most $(1+\epsilon)^k$ in $H \backslash \{x\}$. We now do the analysis of the size of subgraph H.

Lemma 6.9. *With high probability, $|H| = O(k \times n^{1+1/k} \log n \times \log_{1+\epsilon}(nW))$.*

Proof. It is easy to see that for any $i \in [1, k]$, $|S_i| = O(\frac{n^{1+1/k}}{n^{i/k}})$. Consider the collection defined below.

$$\mathcal{C} = \{\sigma(\text{HD}_\alpha(v)) \mid v \in S_i, \alpha \in \text{POWERS}(1+\epsilon)\}$$

The collection $\mathcal{C}$ will contain at most n^2 sets as for any vertex v we can have at most n detours corresponding to n ancestors of v in T. From Lemma 6.3 it follows that for each $\sigma(\text{HD}_\alpha(v))$ of size greater than $n^{i/k} \log n$, w.h.p. the last $n^{i/k} \log n$ vertices of $\sigma(\text{HD}_\alpha(v))$ will contain a

vertex from S_{i+1}. In other words, w.h.p. for each $v \in S_i$ and each $\alpha \in \text{POWERS}(1+\epsilon)$, $\text{SUFFIX}\big(\sigma(\text{HD}_\alpha(v)), S_{i+1}\big)$ is of size at most $n^{i/k}\log n$. So for any vertex v in S_i we add at most $n^{i/k}\log n \times \log_{1+\epsilon}(nW)$ edges in Step 7. Since $|S_i| = O(\frac{n^{1+1/k}}{n^{i/k}})$, and i ranges from 1 to k, w.h.p. H contains $O(k \times n^{1+1/k}\log n \times \log_{1+\epsilon}(nW))$ edges. □

We thus get the following lemma.

Lemma 6.10. *Let G be a directed weighted graph on n vertices with maximum edge weight W and s be the designated source vertex. Then we can compute in polynomial time a subgraph H satisfying the following relation: for any $x, v \in V$, $dist_{H\setminus x}(s,v) \leq (1+\epsilon)^k dist_{G\setminus x}(s,v)$. The size of H is $O\big(kn^{1+1/k}\log(n)\log_{1+\epsilon}(nW)\big)$ with high probability.*

On substituting $\epsilon_{new} = \epsilon_{old}/(2k)$, and $k = \log_2(n)$ in Lemma 6.10 we get the following theorem.

Theorem 6.2. *Let G be a directed weighted graph on n vertices with maximum edge weight W and s be the designated source vertex. Then we can compute in polynomial time a subgraph H satisfying the following relation: for any $x, v \in V$, $dist_{H\setminus x}(s,v) \leq (1+\epsilon) dist_{G\setminus x}(s,v)$. The size of H is $O\big(n\log^3(n)\log_{1+\epsilon}(nW)\big)$ with high probability.*

6.4 Precursor to a Compact Routing

In the previous section, we showed a construction of a sparse subgraph H that contained a $(1+\epsilon, k)$-preserver of each detour in G. In this section we slightly modify the construction of subgraph H which will help us later in obtaining an efficient routing scheme.

Recall that a $(1+\epsilon, k)$-preserver in H is a concatenation of at most k suffixes of detours, that were added to H in the k distinct rounds of Algorithm 6.2. Our routing scheme upon failure of any node x will route packets along these suffixes. In order to efficiently route along a suffix of a $(1+\epsilon, k)$ preserver, we require that each vertex on the suffix knows its successor on the suffix. Now if the frequency of a vertex w in all the suffixes included in H by Algorithm 6.2 is small, then the routing table required at w will be small. Thus as a precursor to routing we require that the frequency of any vertex in the suffixes added to H is bounded. More formally, if $Q_1, ..., Q_t$ are

the suffixes added in a round i of Algorithm 6.2, then we need that the frequency of each vertex in $\{\sigma(Q_1), .., \sigma(Q_t)\}$ is small.

Though Algorithm 6.2 ensures that the average frequency of a vertex is $\widetilde{O}(\log_{1+\epsilon}(n\,W))$, it fails to provide any bound on the maximum frequency. Recall that in any round, say i, of Algorithm 6.2, we compute the next subset S_{i+1} using random sampling. Instead of building this hierarchy of subsets randomly, we present in this section a deterministic algorithm that employs more insight into the structure of the suffixes. We first explain the simpler case when G is a DAG.

6.4.1 Computation of S_{i+1} for acyclic graphs

When G is a DAG, there is a simple greedy algorithm to compute S_{i+1}. Its pseudocode is given in Algorithm 6.3. The input to this algorithm is the collection $\mathcal{C} = \{\sigma(\text{HD}_\alpha(v)) \mid v \in S_i, \alpha \in \text{POWERS}(1+\epsilon)\}$ and a parameter d to be fixed later on. The first step is to sort the vertices of G in the topological ordering and assign S as empty set. Next we repeat the following two steps as long as there exists a vertex whose frequency in $\mathcal{C}$ is greater than d: (i) Pick a vertex w with maximum topological numbering whose frequency is greater than d and add it to set S; (ii) Remove from $\mathcal{C}$ each sequence σ in which w is present. Note that if vertex w leads to the removal of a sequence $\sigma' \in \mathcal{C}$, then the vertices in $\text{SUFFIX}(\sigma', \{w\})$ will have frequency at most d at that moment. This is because vertices in each sequence appear in topological ordering, and among all vertices with frequency greater than d we pick that vertex which has maximum topological numbering. Thus set S satisfies the condition that frequency of each vertex in $\{\text{SUFFIX}(\sigma, S) \mid \sigma \in \mathcal{C}\}$ is bounded by d. Notice that each time a vertex is added to S, at least d sequences are removed from $\mathcal{C}$. Therefore, the size of S is at most $|\mathcal{C}|/d$. The set S_{i+1} is assigned to be the above computed set S.

Algorithm 6.3: GREEDY-CONSTRUCTION($\mathcal{C}, d$)

```
L ← V(G) sorted in topological ordering;
S ← ∅;
while ∃ u ∈ V s.t. FREQ(u, C) > d do
    w ← last vertex in L whose frequency in C is greater than d;
    S ← S ∪ {w};
    Remove all those σ from C that contains vertex w;
end
Return S;
```

Recall that in Algorithm 6.2, $|S_i| \leq n/n^{(i-1)/k}$, and thus $|\mathcal{C}| \leq n/n^{(i-1)/k} \log_{1+\epsilon}(n\,W)$.

By our construction in Algorithm 6.3, we have that $|S_{i+1}|$ is at most $|\mathcal{C}|/d$, which we required to be bounded by $n/n^{i/k}$. To achieve this we assign d to be $n^{1/k}\log_{1+\epsilon}(nW)$. Finally notice that the frequency of each vertex v in the set of all suffixes added during round i is bounded by $d = n^{1/k}\log_{1+\epsilon}(nW)$. We thus have the following lemma.

Lemma 6.11. *Let G be a directed acyclic graph (DAG) on n vertices. Then there exists a construction for sets $S_1, S_2, \ldots, S_k$ in Algorithm 6.2 satisfying the following conditions.*

1. *For any index i, $|S_{i+1}| \leq n/n^{i/k}$.*
2. *For any index i, frequency of each vertex in $\{\text{SUFFIX}(\sigma(HD_\alpha(v)), S_{i+1}) \mid v \in S_i, \alpha \in \text{POWERS}(1+\epsilon)\}$ is at most $n^{1/k}\log_{1+\epsilon}(nW)$.*

6.4.2 Computation of S_{i+1} for general directed graphs

Our algorithm for general graphs can be seen as a combination of the divide-and-conquer and the greedy approach. Given a sequence σ, let FIRST-HALF(σ) and SECOND-HALF(σ) respectively denote the subsequences of σ obtained by splitting it at midpoint. Our first step is to compute collections $\mathcal{C}_1$ and $\mathcal{C}_2$ by splitting each $\sigma \in \mathcal{C}$ at midpoint and adding FIRST-HALF(σ) to $\mathcal{C}_1$ and SECOND-HALF(σ) to $\mathcal{C}_2$. Next we assign S as GREEDY-CONSTRUCTION($\mathcal{C}_2, d\log n$). Since there is no topological ordering, we take L to be just any arbitrary ordering of $V(G)$ in function GREEDY-CONSTRUCTION. It easy to see that set S will be of size at most $|\mathcal{C}|/(d\log n)$.

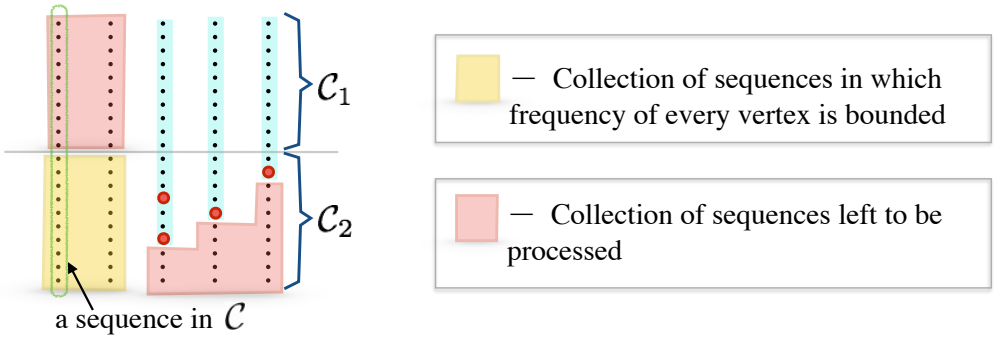


Figure 6.3: Divide and conquer approach: vertices in red color represent the set S.

Consider any sequence $\sigma \in \mathcal{C}$. Let us first consider the case when SECOND-HALF(σ) $\cap\, S$ is non empty. (See Figure 6.3). If SUFFIX(σ, S) has no vertex of large frequency, then we are done with this sequence. However, since G is not acyclic, it is quite possible that SUFFIX(σ, S) may still

have vertices of large frequency. But, in this case, we are left to take care of only SUFFIX(σ, S), whose length is at most $|\sigma|/2$, for the computation of the set S_{i+1}. We now show how to take care of those sequences whose SECOND-HALF does not contain any vertex from S. Let $\mathcal{C}'$ denote the set of all those σ in $\mathcal{C}$ for which SECOND-HALF$(\sigma) \cap S$ is empty. Notice that the frequency of each vertex in the collection $\{\text{SECOND-HALF}(\sigma) \mid \sigma \in \mathcal{C}'\}$ is bounded by $d \log n$. So for any $\sigma \in \mathcal{C}'$, we can safely discard SECOND-HALF(σ) and consider only FIRST-HALF(σ) for the computation of S_{i+1}. Thus, in this case also the length of sequence has been reduced to at most half.

The above discussion motivates us to design a recursive algorithm that performs at most $\log n$ recursions to obtain the desired set S_{i+1}. In Algorithm 6.4 we present the pseudocode of our construction. Since in each recursive call of DAC-CONSTRUCTION, the set S included into S_{i+1} is of size at most $|\mathcal{C}|/(d \log n)$, the size of the set S_{i+1} is at most $|\mathcal{C}|/d$. Also in each recursive call the frequency of vertices increases by at most $d \log n$, thus the set S_{i+1} will satisfy the condition that frequency of each vertex in $\{\text{SUFFIX}(\sigma, S_{i+1}) \mid \sigma \in \mathcal{C}\}$ is bounded by $d \log^2 n$.

Algorithm 6.4: DAC-CONSTRUCTION$(\mathcal{C}, d)$

```
1 C_1 ← {FIRST-HALF(σ) | σ ∈ C};
2 C_2 ← {SECOND-HALF(σ) | σ ∈ C};
3 S ← GREEDY-CONSTRUCTION(C_2, d log n);
4 C_new ← ∅;
5 foreach σ ∈ C do
6 |   if SECOND-HALF(σ) ∩ S ≠ ∅ then add SUFFIX(σ, S) to C_new;
7 |   if SECOND-HALF(σ) ∩ S = ∅ then add FIRST-HALF(σ) to C_new;
8 end
9 Return (S ⋃ DAC-CONSTRUCTION(C_new, d));
```

Recall that in Algorithm 6.2, $|S_i| \leq n/n^{(i-1)/k}$, and thus $|\mathcal{C}| \leq n/n^{(i-1)/k} \log_{1+\epsilon}(nW)$. By our construction in Algorithm 6.4, we will have that $|S_{i+1}|$ is at most $|\mathcal{C}|/d$, which we required to be bounded by $n/n^{i/k}$. This shows that d must be $n^{1/k} \log_{1+\epsilon}(nW)$. Finally notice that the frequency of each vertex v in the set of all suffixes added during round i will be bounded by $d \log^2 n = n^{1/k} \log_{1+\epsilon}(nW) \log^2 n$. We thus have the following lemma.

Lemma 6.12. *Let G be a directed graph on n vertices. There exists a construction for sets $S_1, S_2, \ldots, S_k$ in Algorithm 6.2 satisfying the following conditions.*

1. *For any index i, $|S_{i+1}| \leq n/n^{i/k}$.*

2. *For any index i, frequency of each vertex in* $\{\text{SUFFIX}(\sigma(HD_\alpha(v)), S_{i+1}) \mid v \in S_i, \alpha \in \text{POWERS}(1+\epsilon)\}$ *is at most* $n^{1/k}\log^2 n \log_{1+\epsilon}(nW)$.

6.4.3 A sparse subgraph with bounded in-degree

We show as a corollary that the alternative construction of sets $S_1, \ldots, S_k$ also gives a bound on the in-degree of each vertex in the sparse subgraph H. Notice that in Algorithm 6.2, an incoming edge is added to a vertex w only if $w \in \text{SUFFIX}\big(\sigma(\text{HD}_\alpha(v)), S_{i+1}\big)$, for some v in S_i and $\alpha \in \text{POWERS}(1+\epsilon)$. So the number of incoming edges added to a vertex in round i is exactly equal to frequency of that vertex in $\{\text{SUFFIX}(\sigma(HD_\alpha(v)), S_{i+1}) \mid v \in S_i, \alpha \in \text{POWERS}(1+\epsilon)\}$. Therefore, Lemma 6.11 and Lemma 6.12 respectively give a bound of $kn^{1/k}\log_{1+\epsilon}(nW)$ and $kn^{1/k}\log^2 n \log_{1+\epsilon}(nW)$ on the in-degree of each vertex in the sparse subgraph preserving distances $(1+\epsilon)^k$ approximately. On substituting $\epsilon_{new} = \epsilon_{old}/(2k)$, and $k = \log_2 n$ we get the following theorem.

Theorem 6.3. *Let G be a directed weighted graph on n vertices with maximum edge weight W and s be the designated source vertex. Then in polynomial time we can compute a sparse subgraph H satisfying the following.*

1. *For any $x, v \in V$, $dist_{H\setminus x}(s,v) \leq (1+\epsilon) dist_{G\setminus x}(s,v)$.*

2. *The in-degree of each vertex in graph H is at most $O\big(\log^2(n)\log_{1+\epsilon}(nW)\big)$ if G is acyclic and $O\big(\log^4(n)\log_{1+\epsilon}(nW)\big)$ if G has cycles.*

6.5 An Oracle and a Labeling Scheme

We first describe a fault tolerant oracle for reporting approximate distances from s. Let v be the query vertex and x be the failed vertex. If x is not an ancestor of v, then the shortest path is the tree path $\text{PATH}_T(s,v)$ which remains intact in $G \setminus x$. So let us consider the more interesting case in which x is an ancestor of v. From Lemma 6.2 it follows that one of the shortest paths to v in $G \setminus x$ is of the form - $\text{PATH}_T(s,a)::D::\text{PATH}_T(b,v)$, where D is a detour avoiding x, and its weight is $dist_G(s,v) + wt^*(D)$.

Notice that if α_0 is the smallest power of $(1+\epsilon)$ for which there exists a vertex $b_0 \in \text{PATH}_T(\bar{x}, v)$ with $\text{HD}_{\alpha_0}(b_0)$ starting from an ancestor of x in T, then the value $dist_G(s,v) + \alpha_0$ would be a $(1+\epsilon)$ approximation of $dist_{G\setminus x}(s,v)$. Thus in order to compute an approximation of $dist_{G\setminus x}(s,v)$ we need to compute this α_0 efficiently. We now describe our data structure that accomplishes this task.

For each $\alpha \in \text{POWERS}(1+\epsilon)$, we create a copy of T and denote it by T_α. For each vertex $y \in T_\alpha$, we set the weight of edge $(parent_{T_\alpha}(y), y)$ as $depth_T(\text{FIRST}_\alpha(y))$. Thus the edge weights in any tree T_α are integers in the range $[0, n-1]$. Notice that for any $\alpha \in \text{POWERS}(1+\epsilon)$, there will exist a detour starting from an ancestor of x and terminating at a vertex of $\text{PATH}_T(\bar{x}, v)$ with wt^* at most α if and only if the weight of the minimum weight edge on $\text{PATH}_{T_\alpha}(\bar{x}, v)$ is smaller than $depth_T(x)$. The smallest such α can be easily computed using the Bottleneck Edge Query (BEQ) data structure of Demaine et al. [DLW14] (see Theorem 2.4) for each of the trees T_α.

In Algorithm 6.5 we present the pseudocode for determining approximate $s - v$ distance in $G \setminus x$. Since the BEQ query on a single tree can be answered in $O(1)$ time, the time for querying all the trees will be $O(\log_{1+\epsilon}(n\,W))$. However, instead of linearly checking all the powers of $(1+\epsilon)$ in the increasing order, if we perform a binary search on $\text{POWERS}(1+\epsilon)$, then query time is improved to $O(\log_2 \log_{1+\epsilon}(n\,W))$.

Algorithm 6.5: Determining $(1+\epsilon)$ approximate distance of v from s in graph $G\setminus\{x\}$.

```
if (x is not an ancestor of v) then Return dist_G(s, v);
i ← depth_T(x);
foreach α ∈ POWERS(1 + ε) in increasing order do
    j ← BEQ(x, v, T_α);
    if j < i then Return (dist_G(s, v) + α);
end
Return "Unreachable"
```

Finally notice that the space complexity is $O(n \log_{1+\epsilon}(n\,W))$. The following theorem stems from the above discussion.

Theorem 6.4. *Given a directed graph G on n vertices with maximum edge weight W and a source vertex $s \in V$, it is possible to compute a data structure of $O(n \log_{1+\epsilon}(n\,W))$ size that for any failing vertex x and any destination vertex v, reports a $(1+\epsilon)$ approximation of the distance of v from s in $G \setminus \{x\}$ in $O(\log_2 \log_{1+\epsilon}(n\,W))$ time.*

6.5.1 Compact labeling scheme for Oracle

We now present the labeling scheme for oracle. Let v be a query vertex, and x be the failed vertex. Recall Algorithm 6.5. Our first step is to check whether x is an ancestor of v in T. One simple method to achieve this is to perform the pre-order and the post-order traversal of T, and store the pre-order as well as the post-order numbering of each vertex in its label. Now x will be ancestor of v in T if and only if the pre-order numbering of x is smaller than that of v, and the post order numbering of x is greater than that of v. Next we assign i as $depth_T(x)$, extracting out this is also easy since depth information can also be made part of the label.

Thus the only thing that is left is to obtain a labeling scheme for BEQ problem. In [DLW14], Demaine et al. showed that the BEQ problem on any tree T_α is reducible to LCA problem on a cartesian tree, say $\mathcal{T}_\alpha$, which is related to tree T_α as follows.

1. The vertices of T_α constitute the leaves of $\mathcal{T}_\alpha$.
2. The edges of T_α constitute the internal nodes of $\mathcal{T}_\alpha$.
3. For any two vertices $u, v \in T_\alpha$, the least weight edge on $\textsc{Path}_{T_\alpha}(u, v)$ is the LCA of the leaf nodes corresponding to u and v in $\mathcal{T}_\alpha$.

Hence the bottleneck edge query for any two vertices u and v in T_α can be answered by performing an LCA query for leaves u and v in $\mathcal{T}_\alpha$. However, notice that given the labels of u and v in tree $\mathcal{T}_\alpha$, we are not interested in computing the label of the edge stored at the LCA of u and v, rather we are interested in knowing the weight of the edge stored at the LCA.

Alstrup et al. [AHL14] established a labeling scheme for LCA that given the labels of any two vertices u and v in a tree, returns a predefined label associated with the LCA of u and v in the tree. If each predefined label consist of M-bits, then the labels in this scheme for LCA consists of $O(M \log n_0)$ bits, where n_0 is the number of nodes in the tree. In our case, the number of nodes in $\mathcal{T}_\alpha$ is $O(n)$ only. Also the predefined labels of internal nodes in $\mathcal{T}_\alpha$ stores just the depth value, thus M is $O(\log n)$.

Therefore, label of any vertex v stores: (i) the pre-order and the post-order numbering of v, (ii) the depth of v in T, and (iii) the label of v corresponding to LCA queries in each tree $\mathcal{T}_\alpha$, where $\alpha \in \textsc{powers}(1+\epsilon)$. Hence the label of each vertex consists of $O(\log^2 n \log_{1+\epsilon}(n\,W))$ bits. We thus have the following theorem.

Theorem 6.5. *Given a directed graph G on n vertices with maximum edge weight W and a source vertex $s \in V$ it is possible to compute vertex labels of $O(\log^2 n \log_{1+\epsilon}(nW))$ bits such that for any failing vertex x and any destination vertex v, the $(1+\epsilon)$ approximate distance of v from s in $G \setminus \{x\}$ can be determined by processing the labels associated with v and x only.*

Notice that in the above construction, the predefined label of an internal node of $\mathcal{T}_\alpha$, which corresponds to an edge in T_α, say $(parent_{T_\alpha}(y), y)$, just stores the value $depth_T(\text{FIRST}_\alpha(y))$. However, we can also store in the predefined label other relevant information associated with either the detour $\text{HD}_\alpha(y)$, or a $(1+\epsilon, k)$ preserver of $\text{HD}_\alpha(y)$. Then, the result of Alstrup et al. [AHL14] implies the following lemma, which we would use later in the construction of the routing scheme.

Lemma 6.13. *For each y in V and $\alpha \in \text{POWERS}(1+\epsilon)$, let $\Phi_\alpha(y)$ be the M-bit information associated with a $(1+\epsilon, k)$ preserver of $\text{HD}_\alpha(y)$. Then there exists a labeling scheme with labels of $O(M \log n \log_{1+\epsilon}(nW))$ bits such that given the label of any two vertices $x, v \in V$, where x is an ancestor of v in T, the following information can be retrieved.*

(i) Smallest $\alpha \in \text{POWERS}(1+\epsilon)$ such that there exists a vertex $b \in \text{PATH}_T(\bar{x}, v)$ whose $\text{HD}_\alpha(b)$ starts from an ancestor of x,

(ii) Vertex b stated in (i), and the M-bit information, i.e. $\Phi_\alpha(b)$, associated with the $(1+\epsilon, k)$ preserver of $\text{HD}_\alpha(b)$.

6.6 A Compact Routing Scheme

Let v be a query vertex and x be a failed vertex. Let us consider the case in which x is an ancestor of v in T. Let D be the detour corresponding to the shortest s-v path in $G \setminus x$, and let α be the smallest power of $(1+\epsilon)$ greater than or equal to $wt^*(D)$. So there must exist a vertex $b \in \text{PATH}_T(\bar{x}, v)$ such that $\text{HD}_\alpha(b)$ starts from an ancestor of x. Such a vertex can be easily extracted out using the labels of v and x described in Section 6.5. Since we can not afford to store the information of $\text{HD}_\alpha(t)$ explicitly for each $t \in V$, we take help of a $(1+\epsilon, k)$-preserver of $\text{HD}_\alpha(b)$ to route the packets to destination v.

Recall that a $(1+\epsilon, k)$-preserver of a $\text{HD}_\alpha(b)$ can be seen as a concatenation of at most $\ell(\leq k)$ paths, say $Q_\ell, Q_{\ell-1}, Q_2, Q_1$. (See Figure 6.4 (i)). If $b_{\ell+1}, b_\ell, \ldots, b_2, b_1$ are the endpoints of these paths, then

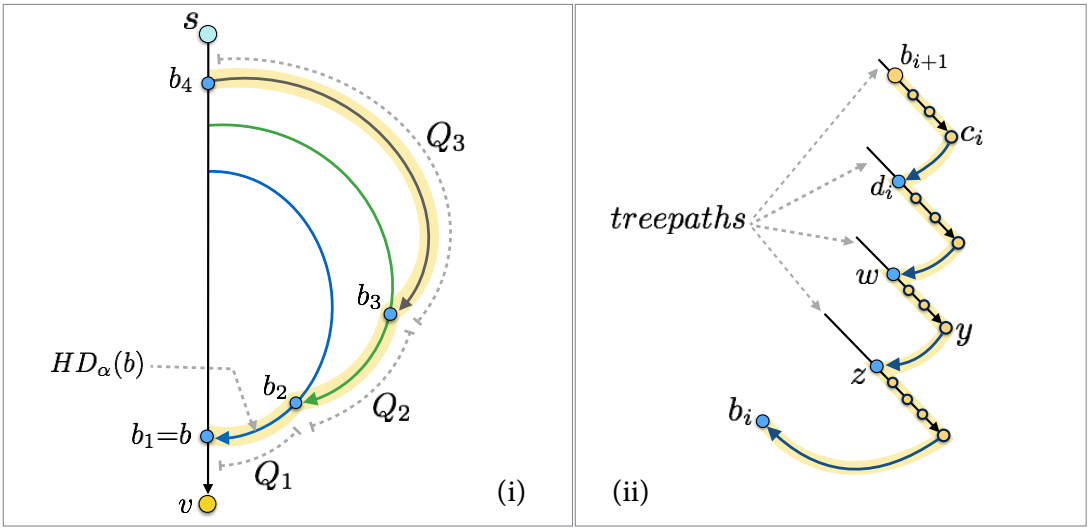


Figure 6.4: (i) Depiction of segments Q_1, Q_2, Q_3 and vertices b_1, b_2, b_3, b_4 in a $(1+\epsilon, 3)$-preserver of $\text{HD}_\alpha(b)$, (ii) A finer description of Q_i showing the first non-tree edge (c_i, d_i).

(i) For each $i < \ell$, Q_i is a suffix of detour of b_i that is added in the i^{th} round of Algorithm 6.2,

(ii) Q_ℓ is a detour of b_ℓ added in the ℓ^{th} round.

Also notice that for each $i \in [1, \ell]$, vertex b_i belongs to set S_i. Now for $i \in [1, \ell]$, let us denote by (c_i, d_i) the first non-tree edge in Q_i. (See Figure 6.4 (ii)). So, d_i is also the first vertex in $\sigma(Q_i)$.

We first extend the labeling scheme described in Section 6.5 to obtain a labeling scheme for routing.

6.6.1 Labeling scheme for routing

The label of each node consists of two subcomponents L_0 and L_1 as described below.

1. In [TZ01], Thorup and Zwick gave a construction of labeling scheme for rooted trees with $O(\log n)$ bit labels such that given the labels of any two nodes w and y (with w being ancestor of y), one can compute the port number of the child of w on $\text{PATH}_T(w, y)$. The first component of the labels, i.e. L_0, would comprise of these labels. This will facilitate easy routing of packets along the tree paths. Also the component L_0 would store the pre-order number, the post-order number, and the depth of vertex in T, so that ancestor descendant relationships can be easily verified. Thus the label L_0 consists of $O(\log n)$ bits.

2. The second component of the labels, that is L_1, will comprise of the labels described in Lemma 6.13, with $\Phi_\alpha(y)$ of a vertex y storing the at most $k+1$ vertices and k non-tree

edges associated with a $(1+\epsilon, k)$ preserver of $\text{HD}_\alpha(y)$, described above. (See Figure 6.4). So given the labels of v and x, we can extract out the value α equal to the smallest power of $(1+\epsilon)$ for which there exists a vertex $b \in \text{PATH}_T(\bar{x}, v)$ with $\text{HD}_\alpha(b)$ starting from an ancestor of x in T. Also we can compute the L_0 label of vertices $b_{\ell+1}, b_\ell, \ldots, b_1$, and edges $(c_\ell, d_\ell), \ldots, (c_1, d_1)$ associated with the $(1+\epsilon, k)$-preserver of $\text{HD}_\alpha(b)$. Finally, notice that $\Phi_\alpha(y)$ consists of $O(k \log n)$ bits of information, thus the label L_1 consists of $O(k \log^2(n) \log_{1+\epsilon}(n\,W))$ bits.

6.6.2 Description of routing table

We now give the construction of the routing table of a vertex $w \in V$. The routing table of w stores: (i) the label $L_0(w)$, and (ii) k segments, where i^{th} segment corresponds to i^{th} round of Algorithm 6.2. Below we describe these k segments.

Let us fix an $i \in [1, k]$. Consider the detour $\text{HD}_\alpha(u)$ for some vertex $u \in S_i$ and an $\alpha \in \text{POWERS}(1+\epsilon)$. Let $u' \in S_{i+1}$ be the last vertex in $\sigma(\text{HD}_\alpha(u)) \cap S_{i+1}$, if it exists, else let u' be $\text{FIRST}_\alpha(u)$. Then the suffix $\text{HD}_\alpha(u)[u', u]$ is added to H in the i^{th} round of Algorithm 6.2. Now let us suppose $w \in \sigma\big(\text{HD}_\alpha(u)[u', u]\big)$ and let (y, z) be the first non-tree edge appearing after w on $\text{HD}_\alpha(u)$. So z is the successor of w in the sequence $\sigma(\text{HD}_\alpha(u))$. Through our routing table we need to ensure that if a packet reaches w then it can easily find the route to vertex z. This route will be just the concatenation $\text{PATH}_T(w, y)::(y, z)$. So corresponding to the triplet (i, u', u) in the routing table of w we need to store the label of the edge (y, z).

Finally notice that in Lemma 6.12 we showed that the frequency of each vertex in the set of all suffixes added in a round is $O(n^{1/k} \log^2(n) \log_{1+\epsilon}(n\,W))$. To store a non-tree edge (y, z), it suffices to store just the labels $L_0(y)$ and $L_0(z)$ that are of $O(\log n)$ bits. Thus the size any of the k segments of the routing table of vertex w will be $O(n^{1/k} \log^3(n) \log_{1+\epsilon}(n\,W))$ bits, and the size of the routing table will be $O(kn^{1/k} \log^3(n) \log_{1+\epsilon}(n\,W))$ bits.

6.6.3 Routing algorithm

We now describe the routing algorithm. Let $j \in [1, \ell]$ be the maximal index such that x is an ancestor of vertices $b_1, \ldots, b_j$. It follows from the proofs of Lemma 6.7 and Lemma 6.8, that the concatenation $P = \langle \text{PATH}_T(s, b_{j+1})::(Q_j, \ldots, Q_1)::\text{PATH}_T(b, v) \rangle$ is a path from s to v in $H \setminus x$

whose length is at most $(1+\epsilon)^k$ times the length of the shortest s-v path in $G\backslash x$. In our routing scheme, the packets from s to v will traverse this path P.

Before starting the routing process we use the labels of x and v to compute the vertices $b_1, \ldots, b_j$ and the edges $(c_1, d_1), \ldots, (c_j, d_j)$, also this information is added to the header of the packet. Now path P can be seen as a sequence of segments of tree paths joined through non-tree edges. So, whenever we reach any vertex $w \in \sigma(P)$, our first step is to calculate the next non-tree edge, say (y, z), appearing after w in P. Once this edge is computed, we traverse the tree path from w to y using L_0 labels. Once we reach y, the packet traverses the edge (y, z). On reaching any vertex $w \in \sigma(P)$, the next non-tree edge, say (y, z), on the path P can be computed as follows.

1. If w is an internal vertex of some Q_i, $i \in [1, j]$, then we can just use the routing table stored at w to find the edge (y, z).

2. If $w = b_i$ for some $1 < i \leq j$, the next non-tree edge will be (c_{i-1}, d_{i-1}) lying in segment Q_{i-1}. This information can be retrieved from the header of the packet itself.

Till now we described that if we are at a vertex w in $\sigma(P)$ then how to navigate to the next vertex in $\sigma(P)$. Notice that (c_j, d_j) is the first non tree edge in P. So to reach the first vertex d_j in $\sigma(P)$, path $\textsc{Path}_T(s, c_j)::(c_j, d_j)$ can be traversed using L_0 labels. Finally when we reach the last vertex in $\sigma(P)$, that is b_1, then again we use L_0 labels to traverse path $\textsc{Path}_T(b_1, v)$ to reach vertex v. The pseudocode of this routing procedure is described in Algorithm 6.6.

Notice that the size of each label is $O(k \log^2(n) \log_{1+\epsilon}(nW))$ bits, and the size of each routing table is $O(kn^{1/k} \log^3(n) \log_{1+\epsilon}(nW))$ bits. Also the size of header attached to the packets is $O(k \log n)$ bits and the stretch achieved is $(1+\epsilon)^k$. On substituting $\epsilon_{new} = \epsilon_{old}/(2k)$ and $k = \log_2(n)$ we get the following theorem.

Theorem 6.6. *For a directed network, there exists a fault tolerant scheme for routing packets from a fixed source vertex s with the following properties.*

1. *The label of each vertex consists of $O(\log^4(n) \log_{1+\epsilon}(nW))$ bits and the routing table consists of $O(\log^5(n) \log_{1+\epsilon}(nW))$ bits.*

2. *While routing, each packet has to have extra $O(\log^2 n)$ bits as a header.*

Algorithm 6.6: Route($w, v, j, \langle b_\ell, .., b_1, c_\ell, .., c_1, d_\ell, .., d_1\rangle$, nextEdge)

```
1  if (w = v) then Return "Packet received";
2  if (j = 0 or w = b_1) then        /* traversal of tree path PATH_T(b_1, v) */
3  |  w_new ← child of w on PATH_T(w, v);
4  |  Route(w_new, v, 0, <>,null);
5  end
6  if (nextEdge = null) then         /* in this case w will lie in σ(Q_j) */
7  |  if (w = b_j) then                    /* transition from Q_j to Q_{j-1} */
8  |  |  nextEdge ← (c_{j-1}, d_{j-1});
9  |  |  j ← j − 1;
10 |  else
11 |  |  nextEdge ← Routing-Table-Look-Up(w, j, b_{j+1}, b_j);
12 |  end
13 end
14 (y, z) ← nextEdge ;            /* next non-tree edge to be followed */
15 if (w ≠ y) then                 /* traversal of tree path PATH_T(w, y) */
16 |  w_new ← child of w on PATH_T(w, y);
17 else                                          /* traversal of edge (y, z) */
18 |  w_new = z;
19 |  nextEdge = null;
20 end
21 Route(w_new, v, j, ⟨b_ℓ, .., b_1, c_ℓ, .., c_1, d_ℓ, .., d_1⟩, nextEdge);
```

3. *To route packets from s to a destination v under failure of a vertex x, s should know labels (identity) of both v and x.*

4. *The route taken by packets have a stretch of at most $(1+\epsilon)$ times that of the shortest path possible in $G \setminus x$.*

6.7 Lower Bounds

Let ϵ, W, n be such that ϵ lies in interval $(0, 1)$, and $W, n > 1$. We first show the construction of a graph G on $O(n)$ vertices with edge weights in range $[1, 2\,W]$ such that its $(1+\epsilon)$-distance preserving subgraph requires at least $\Omega\big(n \times min\{n, \log_{1+\epsilon} W\}\big)$ edges.

Let L, ℓ be integers to be fixed later on. We construct a graph G on $n + L - \ell$ vertices as follows. The vertex set of G is $\{u_{\ell+1}, u_{\ell+2}, \ldots, u_L, v_1, \ldots, v_n\}$, and the edge set of G is the union of the following two sets (see Figure 6.5).

1. $E_1 = \{(u_L, u_{L-1}), \ldots, (u_i, u_{i-1}), \ldots, (u_{\ell+2}, u_{\ell+1})\}$, and

2. $E_2 = \{(u_i, v_j) \mid i \in [\ell+1, L], j \in [1, n]\}$.

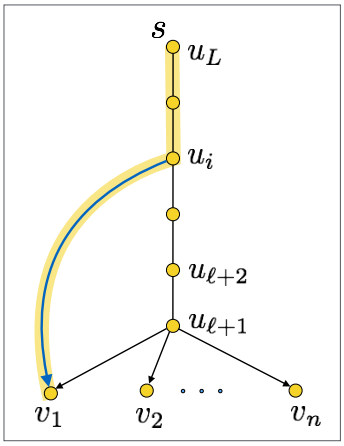


Figure 6.5: The path highlighted in yellow color represents path $P_{i,1}$.

We set $s := u_L$ as the designated source vertex. We now define our weight over the edges of graph G. For each $e \in E_1$, we set $wt(e) = 1$. For each $e \in E_2$, if u_i is the originating vertex of e, then we set $wt(e) = i + (1+2\epsilon)^i$. It is easy to see that the set $E_1 \cup \{(u_{\ell+1}, v_j) \mid j \in [1, n]\}$ constitute the edges of the unique shortest path tree from s.

Now for any $i \in [\ell+1, L]$ and $j \in [1, n]$, let $P_{i,j}$ denote the path $\text{PATH}_T(s, u_i)::(u_i, v_j)$ (see Figure 6.5). Then for any j,

$$\frac{wt(P_{i+1,j})}{wt(P_{i,j})} = \frac{L + (1+2\epsilon)^{i+1}}{L + (1+2\epsilon)^i} > (1+\epsilon), \text{ for } i > \log_{1+2\epsilon}(L) \tag{6.1}$$

We set $L := min\{n, \lfloor \log_{1+2\epsilon} W \rfloor\}$ and $\ell := \lfloor \log_{1+2\epsilon}(L) \rfloor$. Thus G contains at most $2n$ vertices and all edge weights are in the range $[1, 2W]$.

Since i is always greater than $\ell = \lfloor \log_{1+2\epsilon}(L) \rfloor$, from Equation 6.1 it follows that if vertex u_{i-1} fails then the only $(1+\epsilon)$-approximate route to vertex v_j ($j \in [1, n]$) is path $P_{i,j}$. Hence each v_j must keep all its incoming edges in the subgraph preserving approximate distances. There are $L - \ell = \Omega(L) = \Omega\big(min\{n, \log_{1+\epsilon} W\}\big)$ edges for each v_j. Thus, we are able to show that there exists a graph on $O(n)$ vertices with edge weights in range $[1, 2W]$ such that its $(1+\epsilon)$-distance preserving subgraph requires at least $\Omega\big(n \times min\{n, \log_{1+\epsilon} W\}\big)$ edges.

6.7.1 A lower bound of $\Omega(n \log_{1+\epsilon}(W)/\log n)$ on the size of oracle

We now modify the construction of G to prove a lower bound on the size of $(1+\epsilon)$-approximate distance reporting oracle. Let $Z_1, \ldots, Z_n$ be any arbitrary n vectors in $\{0,1\}^{L-\ell}$. We modify the weights of edges lying in the set E_2 as follows. For each $i \in [\ell+1, L]$ and each $j \in [1, n]$, we increase $wt(u_i, v_j)$ to value $(i+W)$, if the $(i-\ell)^{th}$ bit of vector Z_j is one.

Consider failure of a vertex u_{i-1} for some index i, $(\ell+1 < i < L)$. The following two statement holds true.

1. If the $(i-\ell)^{th}$ bit of Z_j is one, then the length of the shortest path from s to v_j in $G \setminus u_{i-1}$ will be at least $L + (1+2\epsilon)^{i+1}$.

2. If the $(i-\ell)^{th}$ bit of Z_j is zero, then path $P_{i,j}$ will be the unique shortest-path from s to v_j in $G \setminus u_{i-1}$. So, in this case the $(1+\epsilon)$-approximate distance of v_j from s in $G \setminus u_{i-1}$ will be at most $(1+\epsilon) \times (L + (1+2\epsilon)^i)$ which is strictly less than $L + (1+2\epsilon)^{i+1}$.

This shows that by querying a $(1+\epsilon)$-approximate distance oracle we can extract out all $(L-\ell)$ bits of arbitrarily chosen vectors $Z_1, \ldots, Z_n$. Thus the oracle must contain at least $n(L-\ell)$ bits or $(n(L-\ell)/\log n)$ words. Hence, we get a lower bound of $\Omega\big(n \times min\{n, \log_{1+\epsilon} W\}/\log n\big)$ on the size of the oracle.

Chapter 7

Conclusion and Open Problems

The main focus of this thesis was on designing fault tolerant data structures for directed graphs.

For the single source reachability problem our main contribution is obtaining a sparse subgraph, referred as k-FTRS, of $O(2^k n)$ size that preserves the reachability information from a designated source vertex after failure of any k edges or vertices. This result directly implies an $O(2^k n)$ size data structure that after any k failures can answer reachability queries (using standard graph search on the k-FTRS avoiding the failed nodes and edges) from source in $O(2^k n)$ time. For the dual failure case, we obtain an $O(n)$ size data structure for answering single source reachability queries in constant time. An open question is whether we can obtain an $O(n)$ size data structure that can answer reachability information from a designated source in sublinear time, for $k(> 2)$ failures. Another seemingly interesting problem is to compute an oracle for answering reachability queries for a fixed source-destination pair upon any k-failures.

We also present an alternative construction for computing dominators, which are closely related to the problem of single fault tolerant reachability. Our construction uses $O(m \log n)$ time and space. The main bottleneck in our result is a data-structure (see Section 2.4.1) that takes $O(m \log n)$ time. It would be interesting to see if this data-structure can be improved to get $O(m)$ time algorithm for computing dominators.

We use our k-FTRS structure to obtain an efficient oracle for reporting strongly connected components of a graph after k failures. The size of our oracle is $O(2^k n^2)$ and the reporting time is $O(2^k n \log^2 n)$. Two of the main building blocks used in this result are heavy path decomposition and the restricted variant of the problem of computing SCCs intersecting a certain path. It is

natural to explore if the ideas used here can help us in obtaining a sparse subgraph as well. That is, can we compute a sparse subgraph H of G such that after any failure of at most k edges/vertices, the strongly connected components of $G \setminus F$ are exactly the same as that of $H \setminus F$?

For the problem of preserving approximate distance from a designated source vertex, we show that for any directed weighted graph we can compute a sparse subgraph of almost linear size that after failure of any edge/vertex preserves distances from the source by a stretch factor of at most $(1 + \epsilon)$. We also obtain efficient oracle, routing scheme, and labeling scheme for this problem. It would be interesting to see if these results can be extended to multiple failures.

Bibliography

[ADD+93] Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.

[AGHP16a] Stephen Alstrup, Cyril Gavoille, Esben Bistrup Halvorsen, and Holger Petersen. Simpler, faster and shorter labels for distances in graphs. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 338–350, 2016.

[AGHP16b] Stephen Alstrup, Inge Li Gørtz, Esben Bistrup Halvorsen, and Ely Porat. Distance labeling schemes for trees. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 132:1–132:16, 2016.

[AHL14] Stephen Alstrup, Esben Bistrup Halvorsen, and Kasper Green Larsen. Near-optimal labeling schemes for nearest common ancestors. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 972–982, 2014.

[BCE05] Béla Bollobás, Don Coppersmith, and Michael Elkin. Sparse distance preservers and additive spanners. *SIAM J. Discrete Math.*, 19(4):1029–1055, 2005.

[BCHR18] Surender Baswana, Keerti Choudhary, Moazzam Hussain, and Liam Roditty. Approximate single source fault tolerant shortest path. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1901–1915, 2018.

[BCPS15] Gilad Braunschvig, Shiri Chechik, David Peleg, and Adam Sealfon. Fault tolerant additive and (μ, α)-spanners. *Theor. Comput. Sci.*, 580:94–100, 2015.

[BCR15] Surender Baswana, Keerti Choudhary, and Liam Roditty. Fault tolerant reachability for directed graphs. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 528–543, 2015.

[BCR16] Surender Baswana, Keerti Choudhary, and Liam Roditty. Fault tolerant subgraph for single source reachability: generic and optimal. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 509–518, 2016.

[BCR17] Surender Baswana, Keerti Choudhary, and Liam Roditty. An efficient strongly connected components algorithm in the fault tolerant model. In *44th International Collo-*

quium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland, pages 72:1–72:15, 2017.

[Ben77] J. L. Bentley. Solutions to Klee's rectangle problems. *Unpublished manuscript, Dept of Comp Sci, Carnegie-Mellon University, Pittsburgh PA*, 1977.

[BGG+15] Davide Bilò, Fabrizio Grandoni, Luciano Gualà, Stefano Leucci, and Guido Proietti. Improved purely additive fault-tolerant spanners. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 167–178, 2015.

[BGK+08] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert Endre Tarjan, and Jeffery Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.*, 38(4):1533–1573, 2008.

[BGLP14] Davide Bilò, Luciano Gualà, Stefano Leucci, and Guido Proietti. Fault-tolerant approximate shortest-path trees. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 137–148, 2014.

[BGLP16a] Davide Bilò, Luciano Gualà, Stefano Leucci, and Guido Proietti. Compact and fast sensitivity oracles for single-source distances. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, pages 13:1–13:14, 2016.

[BGLP16b] Davide Bilò, Luciano Gualà, Stefano Leucci, and Guido Proietti. Multiple-edge-fault-tolerant approximate shortest-path trees. In *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, pages 18:1–18:14, 2016.

[BK13] Surender Baswana and Neelesh Khanna. Approximate shortest paths avoiding a failed vertex: Near optimal data structures for undirected unweighted graphs. *Algorithmica*, 66(1):18–50, 2013.

[BK15] András A. Benczúr and David R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM J. Comput.*, 44(2):290–319, 2015.

[BKMP10] Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. Additive spanners and (alpha, beta)-spanners. *ACM Trans. Algorithms*, 7(1):5, 2010.

[BS07] Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007.

[CCFK17] Shiri Chechik, Sarel Cohen, Amos Fiat, and Haim Kaplan. (1 + epsilon)-approximate f-sensitive distance oracles. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1479–1496, 2017.

[Che13a] Shiri Chechik. Fault-tolerant compact routing schemes for general graphs. *Inf. Comput.*, 222:36–44, 2013.

[Che13b] Shiri Chechik. New additive spanners. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 498–512, 2013.

[Cho16] Keerti Choudhary. An optimal dual fault tolerant reachability oracle. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 130:1–130:13, 2016.

[CLPR10] Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. Fault tolerant spanners for general graphs. *SIAM J. Comput.*, 39(7):3403–3423, 2010.

[CLPR12] Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. f-sensitivity distance oracles and routing schemes. *Algorithmica*, 63(4):861–882, 2012.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[DK11] Michael Dinitz and Robert Krauthgamer. Fault-tolerant spanners: better and simpler. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 169–178, 2011.

[DLW14] Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014.

[DP09] Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *SODA'09: Proceedings of 19th Annual ACM -SIAM Symposium on Discrete Algorithms*, pages 506–515, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[DP10] Ran Duan and Seth Pettie. Connectivity oracles for failure prone graphs. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 465–474, 2010.

[DP17] Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 490–509, 2017.

[DTCR08] Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. Comput.*, 37(5):1299–1318, 2008.

[EP04] Michael Elkin and David Peleg. (1+epsilon, beta)-spanner constructions for general graphs. *SIAM J. Comput.*, 33(3):608–631, 2004.

[FF62] J.R. Ford and D.R. Fullkerson. *Flows in networks*. Princeton University Press, Princeton, 1962.

[FGMT13] Wojciech Fraczak, Loukas Georgiadis, Andrew Miller, and Robert Endre Tarjan. Finding dominators via disjoint set union. *J. Discrete Algorithms*, 23:2–20, 2013.

[FGNW16] Ofer Freedman, Pawel Gawrychowski, Patrick K. Nicholson, and Oren Weimann. Optimal distance labeling schemes for trees. *CoRR*, abs/1608.00212, 2016.

[GIP17] Loukas Georgiadis, Giuseppe F. Italiano, and Nikos Parotsidis. Strong connectivity in directed graphs under failures, with applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1880–1899, 2017.

[GT12] Loukas Georgiadis and Robert Endre Tarjan. Dominators, directed bipolar orders, and independent spanning trees. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, pages 375–386, 2012.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[Moy99] J. Moy. *OSPF: Anatomy of an Internet Routing Protocol.* Addison-Wesley, 1999.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992.

[Par15] Merav Parter. Dual failure resilient BFS structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 481–490, 2015.

[PP13] Merav Parter and David Peleg. Sparse fault-tolerant BFS trees. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 779–790, 2013.

[PS89] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[PT07] Mihai Patrascu and Mikkel Thorup. Planning for fast connectivity updates. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 263–271, 2007.

[PU89a] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(4):740–747, 1989.

[PU89b] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

[Raj12] Varun Rajan. Space efficient edge-fault tolerant routing. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, pages 350–361, 2012.

[ST83] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[TD05] Maxim Teslenko and Elena Dubrova. An efficient algorithm for finding double-vertex dominators in circuit graphs. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 406–411, 2005.

[TZ01] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *SPAA*, pages 1–10, 2001.

[TZ05] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

[Wil86] Robin J Wilson. *Introduction to Graph Theory*. John Wiley & Sons, Inc., New York, NY, USA, 1986.