Space Efficient Edge-Fault Tolerant Routing

Varun Rajan

Department of Computer Science and Engineering, IIT Kanpur, India.
varunrajan09@gmail.com

Abstract

Let G be an undirected weighted graph with n vertices and m edges, and $k \geq 1$ be an integer. We preprocess the graph in $\widetilde{O}(mn)$[1] time, constructing a data structure of size $\widetilde{O}(k \cdot \texttt{deg}(v) + n^{1/k})$ words per vertex $v \in V$, which is then used by our routing scheme to ensure successful routing of packets even in the presence of a single edge fault. The scheme adds only $O(k)$ words of information to the message. Moreover, the stretch of the routing scheme, i.e., the maximum ratio of the cost of the path along which the packet is routed to the cost of the actual shortest path that avoids the fault, is only $O(k^2)$.

Our results match the best known results for routing schemes that do not consider failures, with only the stretch being larger by a small constant factor of $O(k)$. Moreover, a 1963 girth conjecture of Erdős, known to hold for $k = 1, 2, 3$ and 5, implies that $\Omega(n^{1+1/k})$ space is required by any routing scheme that has a stretch less than $2k + 1$. Hence our data structures are essentially space efficient. The algorithms are extremely simple, easy to implement, and with minor modifications, can be used under a centralized setting to efficiently answer distance queries in the presence of faults.

An important component of our routing scheme that may be of independent interest is an algorithm to compute the shortest cycle passing through each edge. As an intermediate result, we show that computing this in a distributed model that stores at each vertex the shortest path tree rooted at that node requires $\Theta(mn)$ message passings in the worst case.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Routing, Fault tolerant algorithms, Space efficiency, Distributed data structures, Tight bounds

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2012.350

1 Introduction

Computer networks are increasingly becoming distributed in nature and consequently recent years have seen extensive work in the areas of distributed algorithms and computing. In the distributed model of computation, which is a more restricted version of the heavily researched parallel model, the processors or vertices, are usually bereft of access to a common shared memory, have only limited knowledge about the network topology and need to communicate with each other through message passing. Evidently this makes the problem of efficiently routing messages of core importance to distributed computing.

The process of routing a packet involves successively passing it through a series of vertices until it reaches its destination. A routing scheme is the underlying algorithm that runs as a background process on all the vertices and manages the routing process. When a vertex receives a message, the routing scheme processes the information stored in the packet header and determines what needs to be done with the message using the routing information

[1] The notation $\widetilde{O}(h(n))$ is short for $O(h(n) \log^{O(1)} n)$

available locally at the vertex. While it is often desirable to route the packet along the shortest path to its destination, its a well known fact that such an algorithm must use at least $\Omega(n)$ space per vertex.

This has lead to extensive research pertaining to compact or sub-quadratic space routing schemes. The two most important factors to consider while designing such schemes are the space requirements per node and the stretch of the routing scheme, that is, the maximum ratio between the length of the path along which the packet is routed to the length of the actual shortest path that the scheme can have for any graph.

After a series of results (refer [2, 9] for a summary of compact routing schemes), Thorup and Zwick[9] came up with some truly remarkable results. They presented near-optimal compact routing schemes that store only $\widetilde{O}(n^{1/k})$ words of information at each vertex and have a stretch of $2k-1$ when using a handshaking procedure by which the source and destination agree on an $o(\log^2 n)$ bit header that is attached to all packets, or an increased stretch of $4k-5$ when not using handshaking.

Peleg and Upfal[8] give the corresponding lower bounds by proving that any routing scheme for general graphs with a stretch $k \geq 1$ must store at least $\Omega(n^{1+1/(2k+4)})$ bits of routing information in the network. Also a 1963 girth conjecture of Erdős[6], known to hold for $k = 1, 2, 3$ and 5, implies that $\Omega(n^{1+1/k})$ space is required by any routing scheme that has a stretch less than $2k+1$. Hence the results of Thorup and Zwick are essentially optimal.

In this paper we study the problem of *fault tolerant routing* which is but a natural extension of the problem of routing and rather practical given that real world networks are quite prone to a variety of faults. Given a positive edge weighted graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, let a message traveling from a source vertex s to its destination t be currently at vertex u and let the next vertex along the shortest path to t be v. In the event of failure of the edge (u, v), a standard routing scheme would be unable to continue routing until the fault has been rectified. Our objective therefore is to design routing schemes that can ensure that the message can still be routed to t along some other path not containing the faulty component, provided such a path exists. To this effect we store some additional information at each vertex during the preprocessing stage of the routing scheme. Upon encountering a failure, this information is added to the message header to facilitate continued routing.

The first significant work that studies the problem of computing the exact paths in the presence of single edge or vertex failures in weighted directed graphs was done by Demetrescu et al. [5]. They present a data structure of $O(n^2 \log n)$ size which can be computed in $O(mn^2)$ time and reports the shortest path avoiding a faulty edge or vertex between any pair of vertices. Bernstein and Karger[1] improved the preprocessing time to $O(mn \log n)$. But neither of these algorithms is distributed in nature, and are therefore unsuitable for routing.

Courcelle and Twigg[11, 4] study the problem of exact routing avoiding a set of known forbidden vertices in weighted graphs with bounded tree width. Given a graph with tree width k and set F of forbidden vertices, they assign labels of size $\widetilde{O}(k^2)$ bits to each vertex and use this information to aid in the routing process. This space requirement would be construed as rather large as even planar graphs have tree-widths of $\widetilde{O}(n^{1/2})$ and hence would require routing tables of size $\widetilde{O}(n)$. Gavoille and Twigg[7] further improved these results to give tables of size $\widetilde{O}(k)$ for routing in planar graphs, thereby matching the corresponding best known results for shortest path routing within logarithmic factors.

Given an undirected weighted graph with edge lengths in the range $[1, W]$, Chechik et al.[3] give an $O(kn^{1+1/k} \cdot \log n \cdot \log(nW))$ space data structure that can route a message along a path that avoids a known set of faulty edges as long as there are no more than

2 faults. This path along which the message is routed has a stretch of $O(k)$. Chechik[2] further improved these results to any arbitrary number of faulty edges F unknown to s, with the corresponding stretch factors being $\widetilde{O}(|F|^3 k)$ using routing tables of size at most $O(kn^{1/k} \cdot \texttt{deg}(v) \cdot \log n \cdot \log(nW))$ at each vertex v. We note that there has been little work, exact or otherwise, to deal with routing in presence of vertex failures in general graphs.

1.1 New results and core techniques

We present simple space efficient data structures that are used by our routing scheme to route packets even in the presence of a single edge fault.

Model of computation. While the routing process utilizes only the routing information stored locally at each node and in the packet's header and hence is distributed in nature, the computation of this information itself is performed during the preprocessing stage in a sequential manner and using quadratic space.

For the routing stage, we use a standard asynchronous distributed model of computation that restricts the size of message headers to $\widetilde{O}(1)$ bits and has no access to a central shared memory.

Basic ideas and techniques. Consider a circle on a plane with a diametral chord intersecting it at points a and b and dividing it into two segments, say $\mathcal{C}_1$ and $\mathcal{C}_2$. The core idea behind our routing scheme is that the shortest path along the circle between any pair of vertices selected from the same segment will not pass through either a or b.

Let vertices u, w lie in $\mathcal{C}_1$ with u closer to a, and likewise let vertices v, x lie in $\mathcal{C}_2$ with v closer to a. Also assume that the shortest path from u to v along the circle passes through a and that (w, x) is an edge. Now if the first edge on the path from u to v were to fail and a message at u needs to be routed to v, we need only route it along the path $u - w - x - v$. This is easily achieved by storing at each vertex, say u, its routing table as well as the edge (w, x) corresponding to each edge (u, v) incident on it.

Our results. Given a network that uses the compact routing scheme of Thorup and Zwick[9] constructed with an integer k, we give data structures that store $O(k \cdot \texttt{deg}(v))$ additional information per vertex $v \in V$ and the corresponding routing algorithm that routes messages along a path of stretch $O(k^2)$ in the presence of a single edge fault.

This routing scheme adds only $O(k)$ words of data to the message header upon encountering a fault and performs only constant amount of computations per node that lies on the path to its destination (except upon encountering the fault). In particular we prove the following theorem.

▶ **Theorem 1.** *Given a positive edge-weighted undirected graph $G(V, E)$ with $|V| = n$, $|E| = m$ and a positive integer k, there is an $\widetilde{O}(mn)$ time constructible routing scheme that stores $\widetilde{O}(k \cdot \texttt{deg}(v) + n^{1/k})$ space per vertex $v \in V$ and can route messages along a path no longer than $O(k^2)$ times the shortest path in the presence of a single edge fault. The packet passed during the routing process has a header of size $O(k + \log n)$ words.*

1.2 Comparison with existing works

Merits of our results. Compared to the other algorithms that deal with compact edge fault tolerant routing, namely of Chechik et al.[3, 2], our compact routing scheme has lower space requirements and comparable stretch, and hence is more applicable in networks which only suffer from occasional failures.

Table 1 Edge fault tolerant routing schemes

	Demetrescu et al. [5]	**Chechik et al. [3]**	**Chechik [2]**	**Our results**
Graph type	Digraph	Undirected	Undirected	Undirected
Stretch	1	$O(k)$	$\widetilde{O}(\lvert F\rvert^3 \cdot k)$	$O(k^2)$
Distributed?	No	No	Yes	Yes
Space required	$\widetilde{O}(n^2)$ overall	$\widetilde{O}(kn^{1+1/k} \cdot \log W)$ overall	$\widetilde{O}(kn^{1/k} \log W\, \texttt{deg}(v))$ per vertex v	$\widetilde{O}(k \cdot \texttt{deg}(v) + n^{1/k}))$ per vertex v
Faults handled	1	2	$\lvert F\rvert$	1

[1] k is the arbitrary integer used by the underlying compact routing scheme
[2] F is the set of faulty edges
[3] W is the ratio of the weights of the heaviest edge to the lightest edge

Our routing schemes can easily be adapted to answer distance queries in presence of faults in $O(k)$ time by using the approximate distance oracles of Thorup and Zwick[10] instead of their compact routing scheme to report distances within $O(k^2)$ times the actual distances.

Demerits of our results. Approximate algorithms are generally considered to be inferior to exact ones but our algorithms do give significant improvements in terms of space requirement over the other results that handle exact failures, particularly the ones in [11, 4, 7].

For the preprocessing stages, we use sequential algorithms, not distributed ones. But as is shown in Corollary 13, even with complete access to the routing tables at each node, the number of messages required to be communicated through the network would still be rather large and undesirable.

Table 1 gives a comprehensive summary of our results and the previous works related to edge fault tolerant routing schemes.

1.3 Organization of the Paper

We describe the preliminary concepts and notations used throughout the paper in Section 2. In Section 3 we present an algorithm for computing the shortest cycle passing through each edge. We also provide a tight bound on the number of messages required to be communicated between the nodes, if the problem were to be solved in a distributed model. In Section 4 we present an $O(k^2)$-approximate routing scheme for handling single edge failures in graphs that perform routing using the compact routing scheme of Thorup and Zwick.

2 Notations and Preliminaries

Without loss of generality we make the following assumptions throughout this paper.

- The vertices in the network know each others' addresses.
- They communicate with each other using packets containing small headers used to store the routing information.
- All shortest paths are unique. We can always break ties by either ranking paths or adding small fractional weights to the edges.
- No more than a single edge failure occurs at any time.
- Message sizes and space bounds are measured in memory words where a single word is sufficient to store the edge or vertex labels and edge weights.

Also we often simply refer to storing information at an edge, when in essence this implies doing so at both of its end-vertices.

Let $G(V,E)$ be the given undirected graph with $|V| = n$, $|E| = m$ and a static weight function $\mathcal{W} : E \rightarrow \mathbb{R}^+$ defined over its edges. Let $s, t \in V$ be the source and destination vertices, respectively and let R be the underlying routing algorithm. We then define the following generic notations.

$\mathcal{T}_u$: the shortest path tree rooted at u.
$\mathcal{T}_u(v)$: the subtree of $\mathcal{T}_u$, rooted at v.
$\mathcal{P}(u,v)$: the original shortest path between u, v.
$\delta(u,v)$: the length of the path $\mathcal{P}(u,v)$.
$\mathcal{F}(u,v)$: the first edge along $\mathcal{P}(u,v)$.
$\mathcal{L}(u,v)$: the last edge along $\mathcal{P}(u,v)$.

We also apply the subscript n to $\mathcal{P}(u,v)$ and $\delta(u,v)$ to refer to the shortest path in the presence of a faulty edge, say e which may be provided as a third argument, unless it is obvious from context. The following are some terminologies and properties that are fundamental to the paper.

▶ **Definition 2.** *Consider a packet being routed along a path, say $\mathcal{P}$ and let a component, say e be faulty. Then the* **stretch** *of $\mathcal{P}$ is defined as the ratio $|\mathcal{P}|$ to $\delta_n(u,v)$. We say the path is* **exact** *if it has a stretch of 1. The stretch of the routing scheme, say k is defined as the maximum stretch that any path in any graph can have under that scheme and such a scheme is said to be a k-approximate routing scheme.*

Stretch and space requirement are indeed the most important factors to consider when designing approximate routing schemes.

▶ **Property 3** (Optimal subpath property). *Any subpath of a shortest path is also a shortest path.*

▶ **Definition 4.** *The* **mid-point** *of a path, say $\mathcal{P}(u,v)$ is defined as that virtual point on $\mathcal{P}(u,v)$ that is equidistant from both u and v. This point may lie on some edge or may itself be a vertex.*

▶ **Definition 5.** *The* **mid-edge** *of a path, say $\mathcal{P}(u,v)$ is defined as*
1. *that edge on $\mathcal{P}(u,v)$ on which its mid-point lies or*
2. *either of the edges on $\mathcal{P}(u,v)$ incident on the vertex on which its mid-point lies.*

Given an edge (u,v) and the shortest cycle through it, say $\mathcal{C}$, we denote the mid-edge of the path $\mathcal{C} \setminus (u,v)$ using the notation $\mathcal{M}(u,v)$. Based on the discussion in the previous section, one may notice that, given access to an exact shortest path oracle, the $\mathcal{M}(u,v)$ values provide a compact means of storing the shortest cycle passing through each edge. An exact routing scheme can then easily use $\mathcal{M}(u,v)$ to route from u to v in the event of failure of (u,v). But the same may not be the case when using an approximate routing scheme. Hence we define separating edges, which are essentially generalizations of mid-edges.

▶ **Definition 6.** *Given a faulty component, say (u,v) and the shortest cycle through it, say $\mathcal{C}$ consider a subset of the edges along the path $\mathcal{C} \setminus e$, given in cyclic order as $(v_1,v_2), (v_3,v_4), \ldots, (v_{2l-1}, v_{2l})$. We say that these edges are* **separating edge(s)** *if the path to be taken from v_{2i} to v_{2i+1} by the underlying routing scheme does not contain (u,v), for all $i \in [0,l]$, where v_0 is u and v_{2l+1} is v.*

Obviously the set of all the edges along the path $\mathcal{C} \setminus e$ is a valid set of separating edges, but storing them would be rather costly. Consequently our objective once the cycles are computed is to find a small set of edges that are valid separating edges. The routing scheme then simply routes from edge e_i to e_{i+1} until the packet reaches v.

Information stored in the packet header. Upon encountering a faulty edge, say e we add its separating edges to the packet header in the form of a queue, say $\mathcal{Q}_{sep}$ sorted by the order that they lie on the shortest cycle of e. We assume that the standard queue operations, namely *enqueue*, *dequeue* and a subroutine, say *makeCopy* to copy one queue to another are available.

In addition to this, the header would also contain information stored by the underlying routing scheme, which would naturally contain the field `destination` to store the destination of the message. We also add a field `rejoinVertex` to store the vertex v at which we rejoin the original path.

3 Computing and Storing Shortest Cycles

In this section, we describe an algorithm for solving the Compute Shortest Cycles problem, that is, the problem of computing the shortest cycle passing through each edge in the given graph. We only require that given access to an all pairs shortest paths (APSP) oracle, any shortest path be reported optimally in time proportional to its length. To this end, we formalize the idea of using mid-edges for efficiently storing the shortest cycles through each edge and give algorithms for computing them. We start by proving the following simple lemma.

▶ **Lemma 7.** *As G is undirected, an edge (u, v) together with $\mathcal{P}_n(u, v)$ forms the shortest cycle passing through (u, v).*

Proof. Suppose that this is not the shortest cycle passing through (u, v). Then the required shortest cycle must be formed by (u, v) and some path, say $\mathcal{P}_x(u, v)$. So $\delta_x(u, v) + \mathcal{W}(u, v) < \delta_n(u, v) + \mathcal{W}(u, v)$ implying that $\delta_x(u, v) < \delta_n(u, v)$. But this is a contradiction as $\mathcal{P}_n(u, v)$ is the shortest path between u and v that does not pass through (u, v). Hence, proved. ◀

Following along similar lines of Lemma 7, it can easily be seen that the shortest path between any two vertices on a given shortest cycle through an edge is fully contained within the cycle. The following lemma utilizes this property to formalize the idea of using mid-edges for computing the shortest cycle.

▶ **Lemma 8.** *Given access to an all pairs shortest path oracle, the shortest cycle passing through each edge can be computed by storing $\mathcal{M}(u, v)$ for each $(u, v) \in E$.*

Proof. Suppose that the edge (u, v) is faulty. Let (w, x) be $\mathcal{M}(u, v)$. Consider the shortest path from u to w. As noted above, this path must either be $\mathcal{P}_n(u, w)$ or $\mathcal{C} \setminus \mathcal{P}_n(u, w)$. Now $\delta_n(u, w) \leq \frac{1}{2}\delta_n(u, v)$ since (w, x) is $\mathcal{M}(u, v)$ and lies on $\mathcal{P}_n(u, v)$. Then $|\mathcal{C}| - |\mathcal{P}_n(u, w)| \geq |\mathcal{C}| - \frac{1}{2}\delta_n(u, v) \geq \mathcal{W}(u, v) + \frac{1}{2}\delta_n(u, v) > \frac{1}{2}\delta_n(u, v)$. Hence $\mathcal{P}_n(u, w)$ is the same as $\mathcal{P}(u, w)$. Similarly we can prove that $\mathcal{P}_n(v, x)$ is the same as $\mathcal{P}(v, x)$, and we can report the shortest cycle through (u, v) as $\mathcal{P}(u, w) \cup (w, x) \cup \mathcal{P}(v, x)$. Hence, proved. ◀

Data structure. As characterized by Lemma 8, we store $\mathcal{M}(u, v)$ for all $(u, v) \in E$. We also store the corresponding $\delta_n(u, v)$ values for obtaining the length of the shortest cycles. Only $O(1)$ words per edge are required for storing these values.

Algorithm. Algorithm SHORTESTCYCLES (See Appendix A.1 for pseudo-code) first computes the APSP oracle using Dijkstra's algorithm and then invokes `computeCycles()` on each edge. For the current edge, say (u, v) it checks for each vertex $w \in V$ if $\mathcal{F}(u, w) \neq \mathcal{F}(v, w)$ and $\mathcal{L}(u, w) \neq \mathcal{L}(v, w)$. If either of these checks fail, the optimal subpath property implies that (u, v) does not form a cycle with the paths $\mathcal{P}(u, w)$, $\mathcal{P}(v, w)$. Upon finding a new cycle, say $\mathcal{C}_1$ the algorithm checks if this cycle is shorter than the currently recorded shortest cycle. As the mid-edge must lie on the longer path amongst $\mathcal{P}(u, w)$ and $\mathcal{P}(v, w)$, The algorithm further checks if the mid-point of the path $\mathcal{C}_1 \setminus (u, v)$ lies on either of the edges adjacent to w, and updates $\delta_n(u, v)$ and $\mathcal{M}(u, v)$ if required.

Simply looping over all the vertices in this manner only considers edges in $\mathcal{T}_u$ and $\mathcal{T}_v$ to be mid-edges. As is shown in Lemma 11, the mid-edge of $\mathcal{P}(u, v)$ need not be present in either of these shortest path trees, and therefore `computeCycles()` cannot guarantee that the shortest cycle is correctly determined. Hence, whenever a new cycle is computed by `computeCycles()`, it invokes the `updateCycles()` procedure which further checks if edge (u, v) can be the mid-edge of the two edges along the cycle that are adjacent to w. Lemma 12 shows that this addition is sufficient to ensure that the correct shortest cycles are computed for each edge when the algorithm terminates. The complete pseudo-code for these procedures is given in Appendix A.1.

▶ Remark. It is worth noting that we can easily loop over the edges in a distributed manner. But before running the `updateCycles()` routine, we will then have to send messages to inform vertex w of the existence of edge (u, v). This would need to be done every time a new cycle is computed, leading to a rather large message complexity of $O(mn)$.

Correctness. We are required to prove that the correct values of $\mathcal{M}(u, v)$ are computed for each edge $(u, v) \in E$ when the SHORTESTCYCLES algorithm terminates. To this effect, we need to ensure that the algorithm loops over every cycle that can be the shortest cycle of some edge before it terminates. As the number of cycles that an edge is a part of can be quite large, often going into the exponentials of n, we need to characterize those cycles which can be a shortest cycle to make the search space tractable.

For simplicity, we say that a cycle is a *candidate shortest cycle* of the given edge if it can be the shortest cycle passing through it, and that such a cycle is *reported* if the algorithm checks to see whether it is shorter than the currently recorded shortest cycle. Naturally when a candidate shortest cycle gets reported, any longer candidate shortest cycles that are not yet reported, cease to be candidates. To reiterate, our objective is to report every candidate shortest cycle. The following observation helps substantially reduce the search space.

▶ **Observation 9.** *From the definition of mid-point, it follows that every path contains at least one mid-edge (two, if the mid-point lies on a vertex). By Lemma 8, we can then express the shortest cycle through each edge $(u, v) \in E$ as an union of two shortest paths and their connecting edges, as $(v, u) \cup \mathcal{P}(u, w) \cup (w, x) \cup \mathcal{P}(x, v)$, where (w, x) is $\mathcal{M}(u, v)$.*

This observation reduces the set of candidate shortest cycles of $(u, v) \in E$ to those cycles $\mathcal{C}$, with $(w, x) \in E$ being the mid-edge of $\mathcal{C} \setminus (u, v)$, that are formed by the concatenation, at both ends, of edge-disjoint shortest paths $\mathcal{P}(u, w)$ and $\mathcal{P}(v, x)$. Thus we need only iterate over every pair of edges to determine the shortest cycles, which is still rather costly. Here we make another crucial observation that given such a cycle $\mathcal{C}$ if the edge (w, x) were to belong to the path $\mathcal{P}(u, x)$ (or to $\mathcal{P}(v, w)$), then we can get sufficient information to determine $\mathcal{C}$ by going over the vertex x (or w) instead of the edge (w, x). This intuitively leads us to consider cycles formed by every edge-vertex pair and characterize those candidate shortest cycles of (u, v) that cannot be determined by this approach.

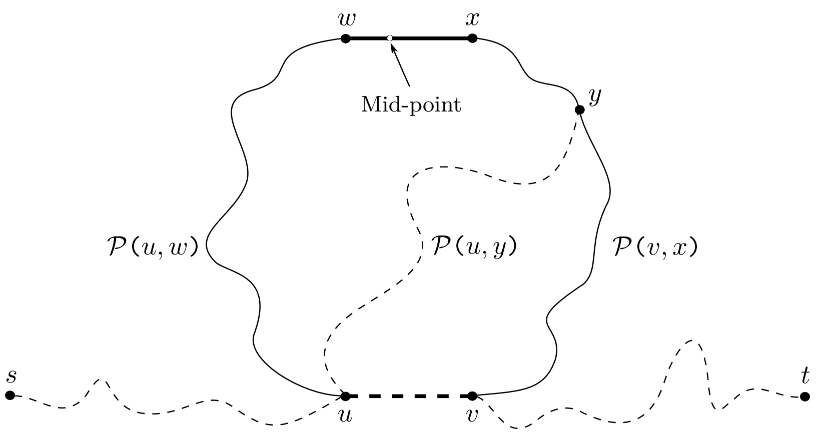

Figure 1 Cycle passing through an edge

For the remainder of this section, we will use the premise that we are given an edge $(u,v) \in E$ and a cycle, say $\mathcal{C}$ passing through it ($\mathcal{C}$ need not be the shortest cycle through it). Without loss of generality, assume that the mid-point of the path $\mathcal{C} \setminus (u,v)$ lies on an edge $(w,x) \in E$. By the definition of mid-edges, it then follows that $\mathcal{P}(u,w)$, (w,x), $\mathcal{P}(x,v)$ and (v,u) together constitute $\mathcal{C}$ (refer fig. 1). Thus, by Observation 9, $\mathcal{C}$ is indeed a representative of all possible candidate shortest cycles. Note that, in the `computeCycles()` procedure, only cycles that are expressed in this manner are checked to see if they are shorter than the currently recorded shortest cycle. Lemma 10 now characterizes the candidate shortest cycles that will be reported by `computeCycles()`.

▶ **Lemma 10.** *If at least one of the paths $\mathcal{P}(u,x)$ and $\mathcal{P}(v,w)$ passes through (w,x), then cycle $\mathcal{C}$ will get reported by* `computeCycles()`*.*

Proof. Assume without loss of generality that path $\mathcal{P}(u,x)$ passes through (w,x). Then by the optimal subpath property, $\mathcal{P}(u,x)$ is the same as $\mathcal{P}(u,w) \cup (w,x)$ and hence does not intersect $\mathcal{P}(v,x)$, implying that cycle $\mathcal{C}$ will be reported by `computeCycles(`u,v`)` when it loops over the vertex x. Hence, proved. ◀

So cycle $\mathcal{C}$ is not reported only if neither $\mathcal{P}(u,x)$ nor $\mathcal{P}(v,w)$ contain (w,x). Lemma 11 further characterizes these candidate shortest cycles of (u,v) that are not reported.

▶ **Lemma 11.** *If cycle $\mathcal{C}$ is not reported by* `computeCycles(`u,v`)`*, then the following statements are equivalent.*

1. *$\mathcal{C}$ is a candidate shortest cycle of (u,v).*
2. *The paths $\mathcal{P}(u,x)$ and $\mathcal{P}(v,w)$ both pass through (u,v).*
3. *w and x belong to different subtrees in both $\mathcal{T}_u$ and $\mathcal{T}_v$.*

Proof. We prove the statements in a cyclic order.

1 ⇒ 2: We prove the contrapositive of this statement which is stated below.

If $\mathcal{C}$ is not reported by `computeCycles()` *and at least one of the paths $\mathcal{P}(u,x)$ and $\mathcal{P}(v,w)$ does not pass through (u,v), then $\mathcal{C}$ is not a candidate shortest cycle of (u,v).*

Suppose $(u,v) \notin \mathcal{P}(u,x)$. As $\mathcal{C}$ is not reported, $(w,x) \notin \mathcal{P}(u,x)$. Then $\mathcal{P}(u,x)$ intersects path $\mathcal{P}(v,x)$ at some vertex, say y that lies on the path from v to x, both inclusive (refer fig. 1). So $\delta(u,y) + \delta(v,y) \leq \delta(u,x) + \delta(v,x) < \delta(u,w) + \mathcal{W}(w,x) + \delta(v,x)$. This shows that $\mathcal{P}(u,y)$, $\mathcal{P}(v,y)$ and (u,v) together form a cycle shorter than $\mathcal{C}$, implying that $\mathcal{C}$ is not a candidate shortest cycle. Similarly we can prove for the case that $(u,v) \notin \mathcal{P}(v,w)$.

2 ⇒ 3: As $\mathcal{P}(u,w)$, (w,x), $\mathcal{P}(x,v)$ and (v,u) together form $\mathcal{C}$ we have $\mathcal{F}(u,w) \neq (u,v)$. Also from statement 2, $\mathcal{F}(u,x) = (u,v)$. So $\mathcal{F}(u,w) \neq \mathcal{F}(u,x)$ implying that w and x belong to different subtrees in $\mathcal{T}_u$. We can similarly prove for $\mathcal{T}_v$.

3 ⇒ 1: From statement 3 we get that edge (w,x) is not present in either $\mathcal{T}_u$ or $\mathcal{T}_v$. This means that `computeCycles(u,v)` never considers the edge (w,x) or consequently, the cycle $\mathcal{C}$. But $\mathcal{C}$ can be shorter than the shortest cycle of (u,v) recorded by it and hence needs to be reported. This concludes the proof as any cycle that needs to be reported is a candidate shortest cycle by definition.

Hence, proved. ◀

Lemmas 10 and 11 thus analyze all the candidate shortest cycles of (u,v) and it turns out that a cycle would still need to be reported for (u,v) if and only if both the path $\mathcal{P}(u,x)$ and $\mathcal{P}(v,w)$ pass through it. All that remains now is to prove that these cycles will get reported by the `updateCycles()` procedure before the algorithm terminates.

▶ **Lemma 12.** *Every cycle which is a candidate shortest cycle is reported, and $\delta_n(u,v)$ and $\mathcal{M}(u,v)$ computed for each edge $(u,v) \in E$ by the algorithm.*

Proof. Suppose that a candidate shortest cycle of (u,v) is not reported. From statement 3 of Lemma 11, we know that w and x belong to different subtrees of $\mathcal{T}_u$, implying that $\mathcal{F}(w,u) \neq \mathcal{F}(x,u)$ and $\mathcal{L}(w,u) \neq \mathcal{L}(x,u)$. But this is the same condition check that is performed when the algorithm loops over the vertex u while computing the shortest cycle through (w,x). Hence the check will pass here, `updateCycles()` procedure will be called with the arguments (u,w,x), and thus the cycle will get reported for (u,v).

The routines used for computing $\delta_n(u,v)$ and $\mathcal{M}(u,v)$ are standard, follow along the same lines as a sequential algorithm for finding the minimum element in an array, and their correctness is easy to see. Hence, proved. ◀

Lemma 11 has an important implication on the efficiency of algorithms that compute shortest cycles in a distributed model that stores at each node the shortest path tree rooted at that node. As each node, say u is unaware of $O(m)$ edges that are not present in $\mathcal{T}_u$ and since the shortest cycle passing through it could be formed together with any of these edges, each of these edges need to inform u of their existence. In the worst case, this would result in $\Omega(mn)$ messages being generated. It is for this reason that we do not perform the preprocessing step using a distributed algorithm. Moreover our algorithm will require $O(mn)$ message passings as described earlier and hence we have the following corollary.

▶ **Corollary 13** (of Lemmas 11, 12). *The* Compute Shortest Cycles *problem requires $\Theta(mn)$ messages passings to compute the shortest cycle through each edge in a distributed model of computing that stores at each node the shortest path tree rooted at that node.*

Time and Space complexities. Computing the APSP oracle requires $O(mn+n^2 \log n)$ time, while the other computations take only $O(mn)$ overall time. Hence the algorithm has an overall time complexity of $O(mn + n^2 \log n)$. We require $O(n^2)$ space for storing the APSP oracle and $O(m)$ space for the output.

Theorem 14 follows directly from the results given in this section.

▶ **Theorem 14.** *Given a graph $G = (V,E)$ with $|V| = n$, $|E| = m$, there is an algorithm that can compute $\delta_n(u,v)$ and $\mathcal{M}(u,v)$ for each $(u,v) \in E$ in $\widetilde{O}(mn)$ steps using $O(n^2)$ space.*

4 An $O(k^2)$-Approximate Edge Fault Tolerant Routing Scheme

In this section we first use the data structure computed in the previous section to develop an algorithm to compute the *separating edges* corresponding to each edge. We then give a simple but space efficient routing scheme that uses this data structure to successfully route packets even in the presence of a single edge fault. While normal routing will be performed using the compact routing schemes of Thorup and Zwick[9], henceforth referred to as the R_{tz} scheme, we will continue to use the APSP oracles for preprocessing. In particular we prove Theorem 1 given in the Introduction section.

Given an integer k, the R_{tz} routing scheme computes tree covers of the given graph, which are a family of induced trees such that, between any pair of vertices in the graph, a path no longer than $4k - 5$ times the shortest path exists in one of these trees. It then stores the tree cover in a distributed manner, using only $\widetilde{O}(n^{1/k})$ space per vertex and uses $o(\log n)$ size headers for routing. We particularly emphasize a feature of this scheme, namely that during the routing process, the message passes through an edge no more than once.

Recall that separating edges of an edge, say (u, v) are edges lying on $\mathcal{P}_n(u, v)$ and given in cyclic order as $e_1, e_2, \ldots, e_l$, such that the path that R_{tz} takes from e_i to e_{i+1} will not contain (u, v), for all $i \in [0, l + 1]$, where $e_0 = e_{l+1} = (v, u)$. We need to compute small sets of separating edges for each edge. The following lemma helps us with this.

▶ **Lemma 15.** *Let $\mathcal{C}_e$ be the shortest cycle through an edge $e \in E$ with $|\mathcal{C}_e| = x$ units. Let $u, v \in \mathcal{C}_e$ be vertices satisfying the conditions $e \notin \mathcal{P}(u, v)$ and $4(k - 1) \cdot \delta(u, v) < x$. Then the path from u to v reported by the R_{tz} scheme does not contain e either.*

Proof. Assume on the contrary that the path reported by the R_{tz} scheme from u to v contains e. As $\mathcal{P}(u, v)$ does not contain e, we get by the optimal subpath property that this path and the reported path together contain a cycle no longer than $\delta(u, v) + (4k-5)\delta(u, v) = 4(k - 1) \cdot \delta(u, v)$ that passes through e. But this implies that a cycle shorter than $\mathcal{C}_e$ exists as $4(k - 1) \cdot \delta(u, v) < x$, giving us a contradiction. Hence, proved. ◀

This lemma essentially implies that we need to divide each shortest cycle into $4(k - 1)$ subpaths separated by single edges that are stored as the separating edges.

Data Structure. We compute and store the separating edges, in cyclic order, as a queue $\mathcal{Q}(u,v)$ for each $(u, v) \in E$. This requires $O(k \cdot \texttt{deg}(v))$ space per vertex v, as shown in Lemma 16. The information required by the R_{tz} scheme is also stored, leading to an overall space requirement of $\widetilde{O}(k \cdot \texttt{deg}(v) + n^{1/k})$ per vertex v.

Preprocessing algorithm. Given a graph $G = (V, E)$, algorithm SEPARATINGEDGES (See Appendix A.2 for pseudo-code) first computes the APSP oracle using Dijkstra's algorithm as before and then runs `computeCycles()` on the graph to compute $\mathcal{M}(u, v)$ and $\delta_n(u, v)$ for each edge $(u, v) \in E$. It then invokes the `compute-kSep()` procedure for each edge to compute separating edges along the shortest cycle passing through it.

Given an edge (u, v) and the corresponding $\delta_n(u, v)$, `compute-kSep()` successively computes the longest section shorter than $\frac{\delta_n(u,v)}{4(k-1)}$ units and adds the next edge to $\mathcal{Q}(u,v)$ as a separating edge. The complete pseudo-code for these procedures is given in Appendix A.2.

Correctness. The correctness of the shortest cycle computing routines have already been proved in the previous section. Procedure `compute-kSep()` trivially ensures that the subpath between two separating edges of (u, v), with indices i and $i + 1$ in $\mathcal{Q}$, are shorter than $\frac{\delta_n(u,v)}{4(k-1)}$ and thus satisfies the requirements implied by Lemma 15. Hence the edges stored in $\mathcal{Q}$ are valid separating edges. Next we bound the size of the queue $\mathcal{Q}$.

▶ **Lemma 16.** *$\mathcal{Q}$ contains only $O(k)$ elements per edge $e \in E$.*

Proof. Given an edge $(u, v) \in E$, each section computed by `compute-kSep()` together with its corresponding separating edge is longer than $\frac{\delta_n(u,v)}{4(k-1)}$. As the path is only $\delta_n(u, v)$ long, there can be no more than $4(k-1)$ separating edges. Hence, proved. ◀

Time and Space complexities. It takes $\widetilde{O}(mn)$ time for computing the APSP oracle and the shortest cycles. For each edge (u, v), `compute-kSep()` traverses the path $\mathcal{P}(u, v)$ performing $O(1)$ computations per vertex along the path. As there can be no more than $n-2$ other vertices along the path, `compute-kSep()` performs $O(mn)$ computations only. Hence the algorithm has an overall time complexity of $\widetilde{O}(mn)$.

The algorithm requires $O(n^2)$ space for storing the APSP oracle, and δ_c and $\mathcal{M}$ require $O(1)$ space per edge. The queues require $O(k \cdot \texttt{deg}(v))$ space per vertex v. Hence the overall space complexity of SEPARATINGEDGES is $O(n^2)$. We have the following theorem.

▶ **Theorem 17.** *Given a graph $G(V, E)$ with $|V| = n$, $|E| = m$, there is an algorithm that can compute the queue of separating edges $\mathcal{Q}(u,v)$ for each edge $(u, v) \in E$ in $\widetilde{O}(mn)$ steps using $O(n^2)$ space. The output, $\mathcal{Q}$ requires $O(k \cdot \texttt{deg}(v))$ space per vertex v, where k is the integer used for constructing the underlying R_{tz} routing scheme.*

4.1 The Routing Scheme

The routing is performed normally using the R_{tz} routing scheme until the faulty edge, say (u, v) is encountered. $\mathcal{Q}(u,v)$ is then copied to $\mathcal{Q}_{sep}$ in the packet header. The routing is then performed by dequeueing an edge, say (w, x) from $\mathcal{Q}_{sep}$ and routing normally to the dequeued edge. This is repeated until the vertex v is encountered. From here normal routing resumes. The complete routing scheme, described in a step-by-step format follows.

1. Route normally until reaching the `destination` or a faulty component, say e.
2. Copy $\mathcal{Q}(e)$ to $\mathcal{Q}_{sep}$ and set v as `rejoinVertex` in the packet header.
3. Dequeue from $\mathcal{Q}_{sep}$ and route to the dequeued edge.
4. Repeat *Step 3* until $\mathcal{Q}_{sep}$ is empty.
5. Route to `rejoinVertex`.
6. Continue normal routing to `destination`.

The correctness of the routing scheme follows directly from the definition of separating edges. Lemma 18 bounds the stretch of this routing scheme.

▶ **Lemma 18.** *The path taken by the routing scheme has a worst-case stretch of $O(k^2)$, where k is the integer used to construct the R_{tz} routing scheme.*

Proof. Consider a packet going from vertex s to t that encounters a faulty edge, say e while traversing along a path no longer than $(4k-5) \cdot \delta(s,t)$. This path together with either the original or the new shortest paths contains a cycle no longer than $(4k-5) \cdot \delta(s,t) + \delta_n(s,t) \leq 4(k-1) \cdot \delta_n(s,t)$. The subpaths between successive separating edges of e along this cycle are further approximated by a factor no more than $4k-5$ as we again use the compact routing scheme. Then the overall distance over which the packet is routed, say X is given as follows.

$$\begin{aligned} X &< (4k-5) \cdot \delta(s,t) + (\text{No. of sections}) * [4(k-1) \cdot \delta(s,t)] \\ &\leq (4k-5) \cdot \delta(s,t) + 4(k-1) * [4(k-1) \cdot \delta(s,t)] \\ &< 16k^2 \delta_n(s,t) \end{aligned}$$

implying that the stretch of the routing scheme is no more than $O(k^2)$. Hence, proved. ◀

Theorem 17 and Lemma 18 together conclude the proof of Theorem 1.

5 Conclusions and Open Problems

The problem of routing in presence of failures has received considerable attention in recent years. We employ an innovative and simple approach to the problem, and present new results that are space-efficient, essentially optimal and match the best known results for compact routing that do not handle failures within a small $O(k)$ factor of stretch. There are several research problems that merit further study.

1. The stretch factor of $O(k^2)$ for our compact routing scheme, may be construed as rather large and undesirable in practice. By properly integrating the routing scheme of Thorup and Zwick rather than using it as a mere black-box, it may very well be possible to significantly reduce the stretch.
2. While the results of Chechik et al.[3, 2] as well as ours give compact space routing schemes for the case of edge failures, such results are not known for the case of vertex faults. Designing compact vertex fault tolerant routing schemes would be quite significant.
3. It would be interesting to see if our techniques can be extended to the case of multiple faults. This work would be in the same vein as that of the multiple fault tolerant scheme of Chechik[2].
4. Extending the algorithms to a streaming/dynamic settings would greatly increase their practicality.

Acknowledgements

We thank the anonymous reviewers for their valuable and detailed comments on improving the presentation.

References

1 Aaron Bernstein and David R. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *STOC*, pages 101–110, 2009.
2 Shiri Chechik. Fault-Tolerant Compact Routing Schemes for General Graphs. In *ICALP (2)*, pages 101–112, 2011.
3 Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. *f*-Sensitivity Distance Oracles and Routing Schemes. In *ESA (1)*, pages 84–96, 2010.
4 Bruno Courcelle and Andrew Twigg. Compact Forbidden-set Routing. In *STACS*, volume 4393 of *LNCS*, pages 37–48, 2007.
5 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.
6 P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice, 1963)*, pages 29–36, 1964.
7 Cyril Gavoille and Andrew Twigg. Compact Forbidden-set Routing on Planar Graphs. Unpublished as yet.
8 David Peleg and Eli Upfal. A Trade-Off between Space and Efficiency for Routing Tables. *J. ACM*, 36(3):510–530, 1989.
9 Mikkel Thorup and Uri Zwick. Compact Routing Schemes. In *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10, 2001.
10 Mikkel Thorup and Uri Zwick. Aproximate Distance Oracles. *J. ACM*, 52:1–24, 2005.
11 Andrew Twigg. *Compact forbidden-set routing.* Tech. Rep. UCAM-CL-TR-678, University of Cambridge, December 2006.