f-Sensitivity Distance Oracles and Routing Schemes

Shiri Chechik [*] Michael Langberg [†] David Peleg [*] Liam Roditty [‡]

June 6, 2011

Abstract

An *f-sensitivity distance oracle* for a weighted undirected graph $G(V, E)$ is a data structure capable of answering restricted distance queries between vertex pairs, i.e., calculating distances on a subgraph avoiding some forbidden edges. This paper presents an efficiently constructible f-sensitivity distance oracle that given a triplet (s, t, F), where s and t are vertices and F is a set of forbidden edges such that $|F| \leq f$, returns an estimate of the distance between s and t in $G(V, E \setminus F)$. For an integer parameter $k \geq 1$, the size of the data structure is $O(fkn^{1+1/k} \log(nW))$, where W is the heaviest edge in G, the stretch (approximation ratio) of the returned distance is $(8k-2)(f+1)$, and the query time is $O(|F| \cdot \log^2 n \cdot \log\log n \cdot \log\log d)$, where d is the distance between s and t in $G(V, E \setminus F)$.

Our result differs from previous ones in two major respects: (1) it is the first to consider *approximate* oracles for general graphs (and thus obtain a succinct data structure); (2) our result holds for an arbitrary number of forbidden edges. In contrast, previous papers concern f-sensitive *exact* distance oracles, which consequently have size $\Omega(n^2)$. Moreover, those oracles support forbidden sets F of size $|F| \leq 2$.

The paper also considers f-sensitive compact routing schemes, namely, routing schemes that avoid a given set of forbidden (or *failed*) edges. It presents a scheme capable of withstanding up to two edge failures. Given a message M destined to t at a source vertex s, in the presence of a forbidden edge set F of size $|F| \leq 2$ (unknown to s), our scheme routes M from s to t in a distributed manner, over a path of length at most $O(k)$ times the length of the optimal path (avoiding F). The total amount of information stored in vertices of G is $O(kn^{1+1/k} \log(nW) \log n)$. To the best of our knowledge, this is the first result obtaining an f-sensitive compact routing scheme for general graphs.

1 Introduction

The problems: This paper considers succinct data structures capable of supporting efficient responses to *distance sensitivity* queries on an undirected graph $G(V, E)$ with edge weights ω. A

[*]Dept. of Computer Science and Applied Math., The Weizmann Institute of Science, Rehovot 76100, Israel, email: {shiri.chechik, david.peleg}@weizmann.ac.il

[†]Computer Science Division, Open University of Israel, 108 Ravutski St., Raanana 43107, Israel, email: mikel@openu.ac.il. Work supported in part by The Open University of Israel's Research Fund (grant no. 46109) and Cisco Collaborative Research Initiative (CCRI).

[‡]Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel, email: liamr@macs.biu.ac.il

distance sensitivity query (s, t, e) requires finding, for a given pair of vertices s and t in V and a *forbidden* edge $e \in E$, the distance (namely, the length of the shortest path) between u and v in $G(V, E \setminus \{e\})$.

An *f-sensitivity distance oracle* is a generalized version of the distance sensitivity data structure, in which instead of a single forbidden edge e, the query may include a set F of size at most f of forbidden edges. In response to a query (s, t, F), the data structure has to return the distance between s and t in $G(V, E \setminus F)$.

An *approximate* distance oracle with stretch k is a data structure that can answer an approximate distance query between two given nodes in a short time. The returned approximate distance is required to be at least the actual distance between the given nodes and at most k times the distance between them.

Combining all the ingredients together, an *f-edge sensitivity approximate distance oracle* with stretch k of the graph G is a data structure that for any edge set $F \subseteq E$ of size at most f can answer an approximate distance queries with a stretch k for the graph $G \setminus F$. In other words, given a set of forbidden edges F such that $|F| \leq f$ and a pair of nodes s and t, the data structure can return in short time an approximate distance d' for the distance $d(s, t, G \setminus F)$ between s and t in $G \setminus F$, where $d(s, t, G \setminus F) \leq d' \leq k \cdot d(s, t, G \setminus F)$. Note that for a possible failure of the set $F \subseteq E$, it could be that the two given nodes are disconnected in $G \setminus F$, in which case the data structure is required to return "infinity".

For certain natural applications in communication networks, one may be interested in more than just the distance between s and t in $G(V, E \setminus F)$. In particular, an *f-sensitivity routing protocol* is a distributed algorithm that, for any set of forbidden (or *failed*) edges F, enables the vertex s to route a message to t along a shortest or near-shortest path in $G(V, E \setminus F)$ in an efficient manner (and without knowing F in advance). In addition to route efficiency, it is desirable to optimize also the amount of memory stored in the routing tables of the nodes, possibly at the cost of lower route efficiency, giving rise to the problem of designing *f-sensitivity* (or *fault-tolerant*) *compact routing schemes.*

The current paper addresses the design of f-sensitivity distance oracles and f-sensitivity compact routing schemes in a relaxed setting in which approximate answers are acceptable. The main results of the paper are summarized by the following theorems. Throughout, our underlying undirected graph G has edge weights $\omega \in [1, W]$, $m = |E|$ and $n = |V|$. For two vertices s and t in G, denote by $\mathbf{dist}(s, t, G \setminus F)$ the distance between s and t in $G(V, E \setminus F)$.

Theorem 1.1 *Let $f, k \geq 1$ be integer parameters. Let $F \subset E$ be a set of forbidden edges, where $|F| \leq f$. There exists a polynomial-time constructible data structure $\mathbf{Sens\_Or}(G, \omega, f, k)$ of size $O(fkn^{1+1/k} \log(nW))$, that on getting a query $s, t \in V$ returns in time $O(|F| \cdot \log^2 n \cdot \log\log n \cdot \log\log d)$ a* distance estimate *$\tilde{d}$ satisfying $d \leq \tilde{d} \leq (8k-2)(f+1) \cdot d$, where $d = \mathbf{dist}(s, t, G \setminus F)$.*

Theorem 1.2 *There exists a 2-sensitive compact routing scheme that given a message M at a source vertex s and a destination t, in the presence of a forbidden edge set F of size at most 2 (unknown to s), routes M from s to t in a distributed manner over a path of length at most $O(k \cdot \mathbf{dist}(s, t, G \setminus F))$. The total amount of information stored in the vertices of G is $O(kn^{1+1/k} \log(nW) \log n)$.*

Related work: In [22], Demetrescu et al. showed that it is possible to preprocess a directed weighted graph in $\tilde{O}(mn^2)$ time[1] to produce a data structure of size $O(n^2 \log n)$ capable of answering 1-sensitivity distance queries (with a forbidden edge or vertex) in $O(1)$ time. The algorithm of [22] also works when the input contains a forbidden *vertex* instead of an edge. In two recent consecutive papers, Karger and Bernstein improved the preprocessing time for 1-sensitivity queries, first to $O(n^2\sqrt{m})$ [14] and later to $\tilde{O}(mn)$ [15]. The size and the query time remain unchanged.

In [26], Duan and Pettie presented an algorithm for 2-sensitivity queries (with 2 forbidden edges or vertices), based on a polynomial time constructible data structure of size $O(n^2 \log^3 n)$ that is capable of answering 2-sensitivity queries in $O(\log n)$ time. The authors of [26] comment that their techniques do not seem to extend beyond forbidden sets of size 2, and that even a solution to the 3-sensitivity problem involving a data structure of size $\tilde{O}(n^2)$ does not seem in reach.

In contrast, the current paper dodges the barrier of [26] and handles forbidden sets F of size greater than 2 by adopting the natural approach of considering *approximate* distances instead of *exact* ones. This approach is used for many "shortest paths" problems. The most notable examples are in efficient computation of approximate "all pairs" distances [10, 38, 28, 3, 25], spanners [46, 47, 18, 11], distance oracles [54, 48, 12] and compact routing schemes [53, 47].

In the approximate setting, when no forbidden edges are considered, distance oracles were introduced by Thorup and Zwick [54]. They have shown that it is possible to preprocess a weighted undirected graph $G(V, E)$ into a data structure of size $O(n^{1+1/k})$ that is capable of answering distance queries in $O(k)$ time, where the *stretch* (multiplicative approximation factor) of the returned distances is at most $2k-1$. As mentioned above, in our work we show that it is possible to preprocess a weighted undirected graph in polynomial time and to produce a data structure of size $O(fkn^{1+1/k} \log(nW))$ that answers f-sensitivity approximate distance queries (s, t, F) for $|F| \leq f$ in $O(|F| \cdot \log^2 n \cdot \log\log n \cdot \log\log d)$ time, where d is the distance between s and t in $G \setminus F$ and W is the heaviest edge in G, and the stretch of the returned distance estimate is $(8k-2)(f+1)$. We assume throughout that edges in G have weight at least 1.

Thus, by considering approximate distances instead of exact ones, our f-sensitivity distance oracle not only solves a more general sensitivity problem but also does it with considerably lower space requirements. In fact, for constant values of k that are significantly larger than the fault parameter f, the stretch of our f-sensitivity distance oracle is the same as in the distance oracles of [54], while its size remains comparable to that of [54].

Very recently, and independently of our work, Khanna and Baswana [39] presented an approximate distance oracle construction for unweighted graphs with a single vertex failure. More precisely, they have shown how to construct a data structure of size $O(kn^{1+1/k}/\epsilon^4)$ that answers an approximate distance query in time $O(k)$ and stretch $(2k-1)(1+\epsilon)$ under a single vertex failure. They have also shown how to find the corresponding approximate shortest path in optimal time. We stress that in this work we consider multiple *edge* faults. Our proof techniques differ significantly from those of [39]. The case of distance oracles that support multiple vertex faults remains open. In [17] we have presented a fault tolerant spanner that supports also vertex failures.

The f-sensitivity distance oracle is closely related to the fundamental problem of dynamic maintenance of all pairs of shortest paths with worst case update time. Demetrescu and Italiano [21],

[1]Here and throughout, the $\tilde{O}(\cdot)$ notation hides logarithmic factors in n.

in a major breakthrough, obtained an algorithm with $\tilde{O}(n^2)$ amortized update time and $O(1)$ query time. Their algorithm was slightly improved by Thorup [51]. In the restricted case of unweighted undirected graphs, Roditty and Zwick [50] showed that for any fixed $\epsilon, \delta > 0$ and every $t \leq m^{1/2-\delta}$, there exists a fully dynamic algorithm with an expected amortized update time of $\tilde{O}(mn/t)$ and worst-case query time of $O(t)$. The stretch of the returned distances is at most $1+\epsilon$. Thorup [52] presented the only non-trivial algorithm with a worst case update time. The cost of each update is $O(n^{2.75})$.

A large gap between the worst case and amortized update times exists also for the problem of *dynamic connectivity* of undirected graphs, where the best worst case update time is $O(\sqrt{n})$ [29] and the best amortized update time is $O(\log^2 n)$ [37].

A connectivity oracle is a data structure that given a set $F \subseteq E$ of size at most f and a pair of nodes s and t can answer in short time if s and t are connected in $G \setminus F$. Pătraşcu and Thorup [42] considered the connectivity problem in a restricted model where all the deleted edges are first deleted in a batch, and queries are answered next. They showed that it is possible to preprocess a given undirected n-vertex graph $G(V, E)$ with m edges in polynomial time and obtain a data structure of size $O(m)$ that allows responding to connectivity queries in $O(f \log^2 n \log\log n)$ worst-case time after a batch of f arbitrary edge deletions.

The design of compact routing schemes has also been studied extensively, focusing on the tradeoffs between the size of the routing tables and the stretch of the resulting routes. For a general overview of this area see [34, 45]. Following a sequence of improvements [47, 4, 7, 20, 27], the first tradeoff between the total size of the routing tables and the stretch of the resulting routing scheme, for general unweighted network topologies, was obtained by Peleg and Upfal [47]. Weighted networks were first considered by Awerbuch et al. [4] who obtained, for every integer $k \geq 1$, a routing scheme that uses $\tilde{O}(n^{1/k})$ space at each vertex, with a stretch factor dependent on k. A better tradeoff was subsequently obtained by Awerbuch and Peleg [7], and efficient schemes for specific values of k were obtained in [20, 27]. The best currently known tradeoffs are due to Thorup and Zwick [53], who present a general routing scheme that uses $\tilde{O}(n^{1/k})$ space at each vertex with a stretch factor of $2k-1$ (using handshaking). Corresponding lower bounds were established in [47, 31, 33, 54].

Fault-tolerant label-based distance oracles and routing schemes for graphs of bounded clique-width are presented in [19]. Given a graph G, a preprocessing stage assigns labels to the vertices of G, so that given the labels of a source vertex s, a destination vertex t, and a set F of forbidden vertices or edges, the scheme efficiently calculates the shortest path between s and t that avoids F (or just its length). For an n-vertex graph of tree-width or clique-width k, the constructed labels are of size $O(k^2 \log^2 n)$. We are unaware of previous results in the literature concerning fault tolerant compact routing schemes for general graphs.

Fault tolerant communication patterns were studied extensively in a variety of contexts, such as data link protocols [35, 30, 1], message transfer over unreliable processors [36], and reliable communication in dynamically changing networks [5, 32, 2]. However, in these studies the goal usually focused on ensuring successful eventual communication, and no attempts were made to optimize the required memory resources or the resulting route lengths.

Fault-tolerance was examined also in the context of other communication patterns, such as broadcast (see, e.g., [44] and the references therein). For a general survey of fault-tolerant broadcasting and gossiping see [43]. For instance, broadcasting with malicious (or Byzantine) transmission failures in the bounded fault model was studied, e.g., in [8, 24], and in the probabilistic failure model, where links or nodes might malfunction at random (independently) on each communication round, in [9, 13, 16]. In the latter papers, failures were assumed permanent and were distributed randomly once for the entire communication process. On the other hand, broadcasting with randomly distributed malicious transmission failures on a line were studied in [40, 44]. Omission transmission failures in the probabilistic model were investigated, e.g., in [23, 44].

Proof techniques: Our results on both f-sensitivity distance oracles and f-sensitivity routing schemes are based at their core on a well known tree cover paradigm used implicitly in [7] and further developed in [6, 18, 49] (see [45]). In this paradigm, given an undirected graph G one constructs a succinct collection of trees that cover the graph G multiple times, once for every different *scale* of distances. The resulting collection of trees has several properties. Primarily, their union acts as a spanner, and thus preserves distances of the original graph up to a multiplicative factor. More important for our constructions, the collection of trees preserves local neighborhoods in an approximate manner as well. Namely, for every vertex v and every distance ρ there is a tree T in the tree cover that includes the entire ρ-neighborhood $B_\rho(v)$ of v, i.e., all vertices of distance ρ from v. Moreover, the path in T between v and any neighbor u in $B_\rho(v)$ is of length proportional to ρ. This last property lends itself naturally to our setting.

For distance oracles, given two vertices s and t, one needs to find the smallest scale factor ρ such that both s and t are in the tree including $B_\rho(s)$. This is done by constructing a connectivity oracle for each tree T, which enables to answer whether s and t are indeed connected in T. For routing schemes, for small values of ρ and upwards, the scheme attempts to route from s to t in the tree containing $B_\rho(s)$; eventually once ρ is approximately the distance between s and t, the routing will succeed. The challenge addressed in this paper is to adapt these ideas to the f-sensitivity setting.

Our construction of f-sensitivity distance oracles enhances the tree cover paradigm in two respects. First, in order to answer queries that involves forbidden edges, one must preprocess each tree into an appropriate connectivity oracle, namely, one that takes forbidden edges into account. To this end, we use a slight variation to the f-sensitivity connectivity oracle proposed recently by Pătraşcu and Thorup [42]. Their oracle maintains a dynamic data structure which is updated for each edge deletion. A connectivity query is answered using the updated data structure. For our construction we need a slightly modified data structure. More precisely, we need a static data structure that given a set $F \subseteq E$ of size at most f and a pair of nodes s and t, can answer if s and t are connected in $G \setminus F$, where the faulty sets F for two consecutive queries could be very different. Therefore, we slightly modify the construction of [42] to get our needs. Given a set $F \subseteq E$ of size at most f and a pair of nodes s and t, we update the data structure of [42] to handle the failure of the set of edges F, we then answer the connectivity query between s and t using the updated data structure. When updating the data structure we "remember" the changes made to the data structure and after answering the query we undo all these changes.

However, this alone does not suffice, as the oracle of [42] requires space which is linear in the number of edges in the graph, whereas we are interested in a data structure that is as small as possible. To reduce the size of the oracle of [42] we use the notion of *connectivity preserver* as will be explained later. Combining these elements with a few additional new ideas leads to the new data structure proposed here.

We now turn to provide a high-level description of our construction of f-sensitivity routing schemes. The challenge at this point lies in identifying additional suitable information to be stored at the vertices of the tree cover that will allow successful routing between a given vertex s and its destination t even if a forbidden edge is encountered. The case of a single forbidden edge is relatively simple. Each edge e in each tree T of the tree cover, if declared as forbidden, disconnects the tree into two connected components. Hence to route from one component to the other, all we need to do is store at the endpoints of e information concerning an alternate *recovery* edge (if such exists) that connects between the components. This will suffice for routing in the 1-sensitivity case, without significantly increasing the stored information. However, it is not hard to verify that this will not suffice once two or more edges may be forbidden. For example, an edge acting as backup for the first forbidden edge may indeed be itself forbidden. A naive solution to this point would involve storing for each edge in T several backup edges, one for any other edge in T. However, this will increase our storage significantly.

Very roughly speaking, to overcome this point, we associate with each edge of T a carefully chosen constant number of backup edges. We then prove, via case analysis, that our choice of backup edges allows successful routing in the presence of 2 edge faults. The crux of our analysis method lies in the structure of our case analysis, which governs the properties needed in the backup edges we choose. It is natural to consider the general case of routing in the presence of f faults for $f \geq 3$. Our current proof techniques do not seem to extend to this case, the difficulty being the structural design of a case analysis allowing to chose a succinct set of backup edges for each edge in T.

The remainder of the paper is organized as follows. In Section 2 we prove Theorem 1.1. In Section 3 we prove Theorem 1.2.

2 f-sensitivity distance oracle

Let $G(V,E)$ be an undirected graph with edge weights ω. We assume throughout that $\omega(e) \in [1,W]$ for every edge e. For an edge set F and two vertices $s,t \in V$, let $\mathbf{dist}(s,t,G \setminus F)$ be the distance between s and t in $G \setminus F (= G(V, E \setminus F))$. In this section we describe the f-sensitivity distance oracle and prove Theorem 1.1.

We start by presenting the construction of the f-sensitivity distance oracle $\mathbf{Sens\_Or}(G,\omega,f,k)$. We then proceed to describe how to answer distance queries of type (s,t,F).

2.1 Constructing the f-sensitivity distance oracle

Construction overview: Our construction is based on a novel combination of three ingredients. The first ingredient is a *tree cover* for the given graph G. The tree cover we use is the skeletal representation of undirected graphs presented in [45, 6, 18]. The second ingredient is the *connectivity oracle* recently presented in [42]. Finally, the third is a simple construction of a sparse subgraph that preserves connectivity with up to f failures. This sparse subgraph was used also in [55, 41]. We start by describing the ingredients that we use. We then present our data structure, which is constructed using a suitable combination of the above three ingredients.

Tree covers: Let $G(V,E)$ be an undirected graph with edge weights ω, and let ρ, k be two integers. Let $B_\rho(v) = \{u \in V \mid \mathbf{dist}(u,v,G) \leq \rho\}$ be the ball of vertices of distance ρ from v. A tree cover $\mathbf{TC}(G, \omega, \rho, k)$ is a collection of rooted trees $\mathcal{T} = \{T_1, \ldots, T_\ell\}$ in G, with root $r(T)$ for every $v \in T$, such that:

(i) For every $v \in V$ there exists a tree $T \in \mathcal{T}$ such that $B_\rho(v) \subseteq T$.

(ii) For every $T \in \mathcal{T}$ and every $v \in T$, $\mathbf{dist}(v, r(T), T) \leq (2k-1) \cdot \rho$.

(iii) For every $v \in V$, the number of trees in $\mathcal{T}$ that contain v is $O(k \cdot n^{1/k})$.

Proposition 2.1 ([6, 18, 45]) *For any ρ and k, one can construct $\mathbf{TC}(\rho, k)$ in time $\tilde{O}(mn^{1/k})$.*

Connectivity oracles: Our second primitive is $\mathbf{Conn\_Or}(G, \omega)$, a connectivity oracle that given a set $F \subset E$ of forbidden edges and a pair of nodes s and t can answer efficiently whether s and t are connected in $G \setminus F$. The properties of the connectivity oracle of [42] are summarized in the following proposition. We use a slight variation of that construction, discussed after the proposition.

Proposition 2.2 ([42]) *There exists a polynomial time constructible data structure $\mathbf{Conn\_Or}(G, \omega)$ of size $O(m)$, that given a set of forbidden edges $F \subset E$ of size f and two vertices $s, t \in V$, replies in time $O(f \log^2 n \log\log n)$ whether s and t are connected in $G \setminus F$.*

Using the data structure presented in [42] "as is" allows us to answer only a single query. That is, in the process of answering a query (s, t, F) the connectivity data structure undergoes certain changes, which prevent us from using it to answer a new query (s', t', F'), asking whether s' and t' are connected in $G \setminus F'$. However, this is a simple technical limitation, caused by the change of the connectivity data structure, and it can be overcome by employing a rollback mechanism that after each query (s, t, F) *rewinds* the changes made to the connectivity data structure until it returns to its original form. This rewinding operation will take time proportional to the query time of the data structure on (s, t, F), and does not effect the original query time stated in [42]. It is now possible to query the structure again using a different set of forbidden edges.

Fault tolerant connectivity preserver: Notice that the size of the data structure $\mathbf{Conn\_Or}$ is $O(m)$. This is necessary in [42] as the size of the forbidden edge set is not known in advance. However, this is not the case in the sensitivity problem, where the size of the forbidden edge set is known in advance. Hence we would like to get a data structure whose size is independent of the number of edges in the graph. To this end, we need to use a sparse representation of the graph G, that has the same connectivity as G itself for any set of f forbidden edges. This is exactly what our last ingredient is used for. We use an edge fault tolerant connectivity preserver $H = \mathbf{Conn\_Pres}(G, \omega, f)$, i.e., a subgraph of $G(V, E)$ such that s and t are connected in $H \setminus F$ iff they are connected in $G \setminus F$ for every two vertices $s, t \in V$ and any subset $F \subseteq E$ of size at most f. Our fault tolerant connectivity preserver has the following properties.

Proposition 2.3 *Let $G(V, E)$ be an undirected graph. There exists a subgraph $H = \mathbf{Conn\_Pres}(G, \omega, f)$ of G of size $O(fn)$ such that for every subset $F \subseteq E$, $|F| \leq f$ and every two vertices $s, t \in V$, s and t are connected in $H \setminus F$ iff they are connected in $G \setminus F$. The subgraph H can be built in time $O(fm)$.*

It was shown in [55, 41] how to construct fault-tolerant connectivity preservers with the desired properties. A closely related problem is the k-edge witness problem studied in [56]. The k-edge witness problem is to preprocess a given graph G so that given a set of k edges S and two nodes u and v, it is possible to answer in a short time whether S is a separator of u and v in G. Roughly speaking, a fault-tolerant connectivity preserver can be constructed by iteratively identifying a spanning forest for G, adding its edges to H, and then removing its edges from G. For completeness, we now describe the construction of the fault-tolerant connectivity preserver. The algorithm, given formally in Figure 1, consists of $f+1$ iterations. Let E_{PR} be the set of edges added to the subgraph so far (initialized to be empty). In each iteration, the algorithm builds a spanning forest for the graph $G \setminus E_{\text{PR}}$. At the end of each iteration we add the edges of the current forest to E_{PR}. After the last iteration, we return $H(V, E_{\text{PR}})$, which is the required fault tolerant connectivity preserver.

```
Algorithm Conn_Pres($G(V,E), f$)

$E_{\text{PR}} \leftarrow \emptyset$
for $i = 1$ to $f+1$ do:
    Construct a spanning forest $F$ for the graph $G \setminus E_{\text{PR}}$
    Add the edges of $F$ to $E_{\text{PR}}$.
Return $H \leftarrow (V, E_{\text{PR}})$
```

Figure 1: Our algorithm for constructing an f-edge fault tolerant connectivity preserver

As the algorithm collects $f+1$ spanning forests, each of at most $n-1$ edges, the total number of edges in the resulting subgraph is $O(fn)$.

We now show that H indeed satisfies the desired properties.

Lemma 2.4 *For every subset $F \subseteq E$, where $|F| \leq f$, and every two nodes $s, t \in V$, if s and t are connected in $G' = (V, E \setminus F)$ then they are also connected in the subgraph $H' = (V, E_{PR} \setminus F)$.*

Proof: Consider a subset $F \subseteq E$, where $|F| \leq f$. Let $H = (V, E_{\text{PR}})$ be the subgraph returned by Algorithm **Conn_Pres**. Consider two nodes s and t connected in G' by a path P. Our goal is to show that s and t are also connected in H'. For that, it suffices to show the following.

Claim: H' contains an alternative path for each edge $e \in E \setminus F$ that is not included in the subgraph H.

To see why the claim suffices, we show how an alternative path for P in H' can be constructed by traversing edges in P or their corresponding detours in H'. Namely, if an edge e of P is also in H (and thus in H', as $e \notin F$), then we traverse e. If $e \notin H$, then we use the alternative path in H' connecting the endpoints of e given by the claim. It follows that there is a path from s to t in H'.

To prove the claim, let H_i be the subgraph added during the ith iteration. Note that the edges of H_i are disjoint for $1 \leq i \leq f+1$. The edge e was not included in each iteration i for $1 \leq i \leq f+1$. Therefore, each H_i contains an alternate path. Hence, H contains $f+1$ disjoint alternative paths for the edge e. As $|F| \leq f$, at most f of those paths are disconnected by the faults in F, so there must be at least one alternative path left in $H' = H \setminus F$. □

Our construction: We now turn to describe our construction of $\mathbf{Sens\_Or}(G, \omega, f, \mathcal{K})$, where $\mathcal{K} = (8k-2)(f+1)$ for integers k and f. (The motivation for using this seemingly odd stretch parameter will become clear shortly.)

Our construction involves $\log(nW)$ iterations, where W is the weight of the heaviest edge in G (hence the diameter of G is at most nW). Each iteration deals with a certain *scale* of distances in the graph G. More specifically, iteration i addresses distances that are at most 2^i in G. Each iteration builds a set of connectivity oracles. Each such oracle will be used to answer connectivity queries on a certain subgraph of G. As we will see shortly, the subgraph for each oracle is specified in two stages, the first defines the vertex set, and the second defines the edge set. We now present our construction for iteration i.

Let H_i be the set of *heavy* edges in G (of weight $\omega(e) > 2^i$). Let G_i be $G \setminus H_i$. It is easy to see that any two vertices that are connected in G by a shortest path of length at most 2^i are still connected in G_i by the same path. For reasons that will become clear shortly, we use the graphs G_i as a base for our construction in iteration i. We start by defining the vertex set of our connectivity oracles. Namely, let $\mathbf{TC}_i = \mathbf{TC}(G_i, \omega, 2^i, k)$. For each tree $T \in \mathbf{TC}_i$ we build a connectivity oracle on the vertices $V(T)$ of T. This completes our first stage.

For the second stage, we define the edges to be considered in the connectivity oracle corresponding to $T \in \mathbf{TC}_i$. Let $G_i|_T$ be the subgraph of G_i induced on the vertices of T. Constructing the connectivity oracle on the edges of $G_i|_T$ actually suffices for our construction. However, as the connectivity oracle uses space that is linear in the number of edges, it is too costly to use it directly on $G_i|_T$. Thus to save space, we consider a sparse representation of $G_i|_T$ that still satisfies our needs. This sparse representation is exactly the fault tolerant connectivity preserver discussed above. Namely, let $\mathbf{Conn\_Pres}_T = \mathbf{Conn\_Pres}(G_i|_T, \omega, f)$ be a fault tolerant connectivity preserver for $G_i|_T$. The subgraph $\mathbf{Conn\_Pres}_T$ is what we use for the connectivity oracle that corresponds to T.

Our data structure includes a connectivity oracle $\mathbf{Conn\_Or}(\mathbf{Conn\_Pres}_T, \omega)$, or simply $\mathbf{Conn\_Or}_T$, for each $T \in \mathbf{TC}_i$. In addition, for each $v \in V$ we compute and store a pointer to the tree $T_i(v) \in \mathbf{TC}_i$ containing $B_{2^i}(v)$. This completes our construction for iteration i.

Lemma 2.5 *The structure* $\mathbf{Sens\_Or}(G, \omega, f, \mathcal{K})$ *is of size* $O(fkn^{1+1/k}\log(nW))$.

Proof: For a tree T, the size of $\mathbf{Conn\_Pres}_T$ is $O(f|V(T)|)$. Therefore the size of $\mathbf{Conn\_Or}_T$ is also $O(f|V(T)|)$. We get that the total size stored for all the connectivity oracles is

$$\sum_{i=1}^{\log(nW)} \sum_{T \in \mathbf{TC}_i} O(f|V(T)|) \;=\; f \sum_{i=1}^{\log(nW)} O(k \cdot n^{1+1/k}) \;=\; O(fkn^{1+1/k}\log(nW))\;.$$

In addition, for each node v and for each $1 \le i \le \log(nW)$, we store a pointer to the tree $T_i(v)$, which can be bounded by $O(n\log(nW))$. □

2.2 Answering queries using the f-sensitivity distance oracle

Answering queries: Given a query (s, t, F) to our data structure, the oracle operates as follows. For each i from 1 to $\log(nW)$, it checks if s is connected to t in the induced graph

```
Algorithm Query($s, t, F$)

For $i \leftarrow 1$ to $\log(nW)$
    if $\mathbf{Conn\_Q}_{T_i(s)}(s, t, F) = \mathit{true}$, then return $\mathcal{K} \cdot 2^{i-1}$
Return $\infty$
```

Figure 2: Algorithm for answering an f-sensitivity distance query

$G_i|_{T_i(s)}$ after the set F of forbidden edges is excluded from it. Recall that G_i is the subgraph of G containing all edges of weight at most 2^i. This is done by querying the connectivity oracle $\mathbf{Conn\_Or}_{T_i(s)}$ of $T_i(s)$ with (s, t, F). Recall that $T_i(s)$ contains all vertices in $B_{2^i}(s)$. If s and t are connected, the oracle returns the value $\mathcal{K} \cdot 2^{i-1}$, otherwise it proceeds to the next i value. If no such i exists, it returns ∞. The formal code is given in Figure 2.

We now show that given a query, the oracle indeed returns a distance estimate that is within a multiplicative factor of $\mathcal{K}$ from $\mathbf{dist}(s, t, G \setminus F)$. Theorem 1.1 then follows by the lemmas below.

Lemma 2.6 (Correctness) *The f-sensitivity distance query algorithm returns an estimate that is within a multiplicative factor of $\mathcal{K}$ from* $\mathbf{dist}(s, t, G \setminus F)$.

Proof: We prove the correctness of the query algorithm in two stages. First, we show that if $d = \mathbf{dist}(s, t, G \setminus F)$ and $i = \lceil \log d \rceil$, then $\mathbf{Conn\_Q}_{T_i(s)}(s, t, F)$ returns true. Next, we show that if i is the first iteration in which $\mathbf{Conn\_Q}_{T_i(s)}(s, t, F)$ is true, then indeed there is a path between s and t in $G \setminus F$ whose length is at most $\mathcal{K} \cdot 2^{i-1}$.

Let P be a shortest path that connects s and t in $G \setminus F$. Note that $2^{i-1} < d = |P| \leq 2^i$. By our definitions, this implies that P is also included in G_i. Moreover, as we define $T_i(s)$ to contain the ball of radius 2^i centered at s, it follows that all the vertices of P belong to $T_i(s)$. Hence, P is included in $G_i|_{T_i(s)} \setminus F$.

As P is a path in $G_i|_{T_i(s)} \setminus F$, it holds that s and t are connected in $G_i|_{T_i(s)} \setminus F$. The subgraph $\mathbf{Conn\_Pres}_{T_i(s)}$ is a fault tolerant connectivity preserver, that is, the graph $\mathbf{Conn\_Pres}_{T_i(s)} \setminus F$ preserves the connectivity information of $G_i|_{T_i(s)} \setminus F$. The actual connectivity query is handled by issuing the query (s, t, F) to the connectivity oracle $\mathbf{Conn\_Or}_{T_i(s)}$. Since s and t are connected in $\mathbf{Conn\_Pres}_{T_i(s)} \setminus F$, the connectivity oracle must return a positive answer indicating that s and t are connected in $G_i|_{T_i(s)} \setminus F$. The returned distance estimate is $\mathcal{K} \cdot 2^{i-1}$. As $|P| > 2^{i-1}$, the stretch of the distance estimate is at most $\mathcal{K}$.

We now turn to the second stage. Notice that since we are using a connectivity oracle on top of a tree cover, it might be that a positive indication is returned before reaching the ith iteration, which contains the shortest path. We therefore need to show that if $j < i$ is the first iteration in which $\mathbf{Conn\_Q}_{T_j(s)}(s, t, F)$ is true, then indeed there is a path in $G \setminus F$ between s and t whose length is at most $\mathcal{K} \cdot 2^{j-1}$. Let $T = T_j(s) \setminus F$. It is easy to see that T is a forest with at most $f + 1$ trees. Let $T_1, \ldots, T_{f+1}$ be the trees of this forest. The depth of $T_j(s)$ is at most $(2k - 1) \cdot 2^j$, hence, the same holds for the trees $T_1, \ldots, T_{f+1}$. From the connectivity oracle we know that s and t are connected in $G_j|_{T_j(s)} \setminus F$. Let P be a path between s and t in $G_j|_{T_j(s)} \setminus F$. We now show that there indeed exists a path (that may differ from P) between s and t in $G_j|_{T_j(s)} \setminus F$ of length at most $\mathcal{K} \cdot 2^{j-1}$.

We decompose the path P into at most $f + 1$ subpaths in the following way. Let $x_1 = s$ and assume that $x_1 \in T_r$, for some $T_r \in \{T_1, T_2, \ldots, T_{f+1}\}$. We define y_1 to be the farthest vertex from x_1 of P that is still in T_r. The vertex x_2 is the first vertex after y_1 in P, and y_2 is again the farthest vertex (in the direction of t) from x_2 of P that is in the same tree as x_2. We continue in a similar manner until we reach t. We denote by $P(x_r, y_r)$ the subpath of P between x_r and y_r. We get that

$$P = P(x_1, y_1) \cdot (y_1, x_2) \cdot P(x_2, y_2) \cdots (y_{\ell-1}, x_\ell) \cdot P(x_\ell, y_\ell) ,$$

where $s = x_1$ and $t = y_\ell$. In order to bound the length of P, let us create an alternate path P' by replacing every subpath $P(x_r, y_r)$ of P with the shortest path between x_r and y_r in the tree that contains them both. Figure 3 illustrates the alternate path P'. It holds that P' is also in $G_j|_{T_j(s)} \setminus F$. Notice that any subpath $P(x_r, y_r)$ is replaced by a path of length at most $2 \cdot (2k-1) \cdot 2^j$. In addition, as P is a path in $G_j|_{T_j(s)} \setminus F$, each edge of P is of weight at most 2^j. This implies that the path P' is of length at most $2(f+1)(2k-1) \cdot 2^j + f \cdot 2^j \leq (4k-1) \cdot (f+1) \cdot 2^j$. Since the shortest path is of length at least 2^{i-1}, where $i > j$, we get a stretch of $\mathcal{K} = (8k-2) \cdot (f+1)$, as required. □

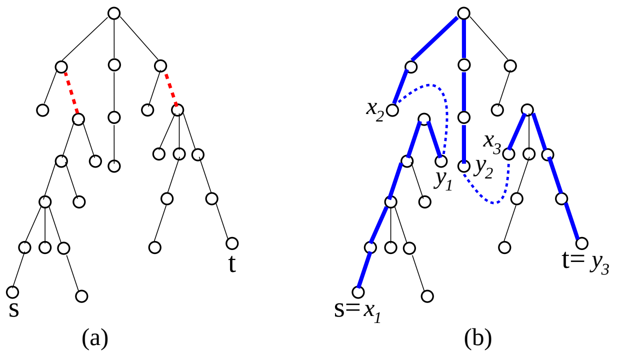

Figure 3: (a) A tree $T_i(s)$ of depth at most $(2k-1)2^i$, where the broken red edges fail. After the failure of the two broken edges, there are three connected subtrees, of depth at most $(2k-1)2^i$ each. (b) The emphasized path represents the modified path P'.

Next, we analyze the running time of the query procedure.

Lemma 2.7 *The f-sensitivity distance query (s, t, F) can be implemented to return a distance estimate in time $O(|F| \cdot \log^2 n \cdot \log\log n \cdot \log\log d)$, where $d = \mathbf{dist}(s, t, G \setminus F)$.*

Proof: In each iteration of the query process, the most time-consuming step is invoking the connectivity oracle, which takes $O(|F| \cdot \log^2 n \cdot \log\log n)$ time. The query algorithm can stop on the first iteration in which the connectivity oracle returns true. Hence, in a straightforward implementation of the query algorithm, as given in Figure 2, there will be at most $O(\log d)$ iterations. The total running time in this case is $O(|F| \cdot \log^2 n \cdot \log\log n \cdot \log d)$. It is possible, however, to reduce the query time to $O(|F| \cdot \log^2 n \cdot \log\log n \cdot \log\log d)$ as follows.
Let i_0 be the first iteration on which the connectivity oracle returns true and let $j = \lceil \log d \rceil$. By the definition of the tree cover used by our construction, it follows that for every iteration after the jth iteration, the connectivity oracle query must also return true. Hence, instead of looking for the first iteration in which the connectivity oracle returns true, we can perform a binary search for an index k such that the connectivity oracle returns true for iteration k and false for iteration $k-1$.

To perform our binary search efficiently, we first need to find an upper bound on i_0, the value of the first iteration on which the connectivity oracle returns true. This can be done by simply checking if $\mathbf{Conn\_Q}_{T_i(s)}(s,t,F)$ is true for $i = 1,2,4,8,\ldots$. As our query is true on any index larger than $j = \lceil \log d \rceil$, we find an index i for which $\mathbf{Conn\_Q}_{T_i(s)}(s,t,F)$ is true in at most $\log j$ iterations. It is not hard to verify that the index i found in this way satisfies $i_0 \le i \le 2j$. Thus our binary search involves at most $O(\log j)$ iterations, yielding a total running time of $O(|F| \cdot \log^2 n \cdot \log\log n \cdot \log\log d)$. $\square$

Combining Lemmas 2.5, 2.6, and 2.7 with a slight change of parameters concludes the proof of Theorem 1.1.

3 2-sensitive compact routing schemes

In this section we present an f-sensitive routing scheme for the case of two forbidden edges (i.e., $f = 2$) and prove Theorem 1.2.

Let $s,t \in V$ and assume that a message is to be routed from s to t. Throughout this section, assume the forbidden edge set F contains two edges, denoted by e_1 and e_2. Loosely speaking, the routing process we suggest is similar in nature to the f-sensitivity query process described in Section 2. That is, our routing scheme involves at most $\lceil \log (nW) \rceil$ iterations. In each iteration i, an attempt is made to route the message from s to t in the graph $G_i|_{T_i(t)} \setminus F$ using the tree $T_i(t)$ augmented with some additional information to be specified below. (Note that in each iteration i, the routing attempt is made on the tree $T_i(t)$ instead of $T_i(s)$; the reason for this will be made clearer later on.) If the routing is unsuccessful, the scheme proceeds to the next iteration. The routing process ends either when the message reaches its destination or after a failure in the final iteration.

Let P be a shortest path between s and t in $G \setminus F$, and let $i = \lceil \log |P| \rceil$. As argued before, P is included in $G_i|_{T_i(t)} \setminus F$. To prove that our routing scheme succeeds, it suffices to prove that it finds a path of length proportional to $|P|$ when the routing is done on the augmented tree $T_i(t)$. Throughout this section, any standard tree routing operation is preformed by using the tree routing scheme of Thorup and Zwick [53]. That scheme uses $(1+o(1))\log_2 n$-bit label for each node. These labels are the only information required for their routing scheme and no other data is stored. In addition, the routing decision at each node takes only constant time.

Note that the routing scheme of [53] may assign a node t a different label $L_T(t)$ for each tree $T \in TC_i$ it belongs to. In order to enable a node s to route a message to a node t over some tree T, it should be familiar with the label $L_T(t)$. Naively, for each node t we could concatenate all labels assigned to t in all trees $T \in TC_i$ it participates in, and use the concatenated string as the new label of t. However, this could lead to prohibitively large labels. Therefore, for each node t, we concatenate only the labels given to t for the trees $T_i(t)$ for $1 \le i \le \log (nW)$ (and some indication on which tree is $T_i(t)$). Therefore, the attempts to route from s to t are made over the trees $T_i(t)$ instead of $T_i(s)$. The size of each node label is $O(\log (nW) \cdot \log n)$. In addition, for every tree T and every node $v \in T$, v stores the original label $L_T(v)$. Each such label is of size $O(\log n)$, therefore, we get an additional $O(\log n)$ factor on the amount of information stored in the vertices. Now consider iteration i where the node s tries to route a message to t over $T_i(t)$. Then s first checks if it belongs to $T_i(t)$; if not, then it proceeds to the next iteration, and so on. In what follows, we assume that the first successful i equals $\lceil \log |P| \rceil$, where P is the shortest path connecting s and t in $G \setminus F$. However, as in the previous section, it might be the case that $s \in T_j(t)$ for $j < i$. This situation is only in our favor, and follows the exact same analysis presented below.

3.1 Warm up: 1-sensitive routing

As a first step, we present a simple routing scheme that allows routing in the case of a single forbidden edge. Let $i \in \{1, \ldots, \log(nW)\}$. Consider the tree cover of Section 2 and let $T \in \mathbf{TC}_i$. We first specify the information stored in the routing table of vertices in T. We then specify our routing procedure.

Each edge $e = (u, v)$ in T, if declared as forbidden, disconnects the tree into two connected components. Let $T_u(e)$ (respectively, $T_v(e)$) be the component that contains u (resp., v). As the route may need to cross from one component to the other, our data structure needs to store at each such edge e some additional information that will allow this task. Specifically, a *recovery edge* of e is any edge e' of G that connects $T_u(e)$ and $T_v(e)$. Let $rec(e)$ be an arbitrary recovery edge of e. If such an edge does not exist, then $rec(e)$ is undefined. The edge $rec(e)$ is stored in both nodes u and v, where node u stores the edge $rec(e)$ together with the endpoint of $rec(e)$ that's in $T_u(e)$ and node v stores $rec(e)$ together with the endpoint of $rec(e)$ that's in $T_v(e)$.

We now show how to route a message from s to t when the edge $e = (u, v) \in E$ is forbidden for use. Let P be a shortest path between s and t in $G \setminus \{e\}$. Let $i = \lceil \log |P| \rceil$, and let $T = T_i(t)$. It suffices to present a scheme that enables routing between s and t in T using the auxiliary information specified above. Initially, the scheme attempts to route the message from s to t using the edges of T alone. If e is not on the routing path, then the message reaches its destination and we are done. Otherwise, $e = (u, v)$ is on this routing path and w.l.o.g. assume that the message encounters the forbidden edge while at u. At this point, the scheme uses the information stored at u. First, notice that P is included in $G_i|_T \setminus \{e\}$ and thus there exists a recovery edge $rec(e)$ in $G_i|_T$. By our construction, u has stored the endpoint of $rec(e)$ in the connected component $T_u(e)$ of u. Thus, u can now route the message towards this endpoint. Then the message can use $rec(e)$ to cross from $T_u(e)$ to $T_v(e)$ and finally the message can continue on its original route to t.

The depth of T is at most $2(k-1)2^i$, therefore the depths of both $T_u(e)$ and $T_v(e)$ are also at most $2(k-1)2^i$. In our routing scheme, starting from s we will first route to u, which may take at most $4(k-1)2^i$ steps (twice the depth). Then from u we route to the endpoint of $rec(e)$ in the connected component $T_u(e)$, which can also be done along a path of length at most $4(k-1)2^i$. The message then crosses from to $T_v(e)$ using $rec(e)$. Note that $rec(e) \in G_i|_T$ is of weight at most 2^i. The routing is continued in $T_v(e)$, and again the path obtained from routing in $T_v(e)$ is of length at most $4(k-1)2^i$. It follows that the entire route is of total length $(12(k-1)+1) \cdot 2^i$.

As already mentioned before, our routing scheme may involve at most $\log(nW)$ iterations. In each failed iteration j, it may also need to inform s about this failure, therefore the length of the path obtained in such an iteration should be multiplied by 2. Consider the first iteration i where the routing succeeds. The total length of the path is at most

$$\sum_{j=1}^{i-1} 2 \cdot (12k-11) \cdot 2^j + (12k-11) \cdot 2^i \;\leq\; 3 \cdot (12k-11) \cdot 2^i \;=\; (36k-33) \cdot 2^i \;.$$

As the original distance from s to t in $G \setminus \{e\}$ is at least 2^{i-1}, the total length of the path is at most $(72k - 66)\mathbf{dist}(s, t, G \setminus \{e\})$.

Theorem 3.1 *There exists a* 1*-sensitive compact routing scheme that given a message* M *at the source vertex* s *and a destination* t*, in the presence of a forbidden edge* e *(unknown to* s*), routes* M *from* s *to* t *in a distributed manner over a path of length at most* $(72k-66)\mathbf{dist}(s, t, G \setminus \{e\})$*. The total amount of information stored in vertices of* G *is* $O(kn^{1+1/k} \log(nW) \log n)$*.*

3.2 2-sensitivity routing

We now turn to describe our 2-sensitivity routing scheme. Let $T \in \mathbf{TC}_i$, where $i \in \{1, \ldots, \lceil \log nW \rceil\}$. Each edge $e = (u, v) \in T$, if declared as forbidden, disconnects the tree into two connected components. Let $T_u(e)$ (respectively, $T_v(e)$) be the component that contains u (resp., v). As the route may need to cross from one component to the other, our data structure needs to store at each such edge e some additional information that will allow this task. Specifically, a *recovery edge* of e is any edge e' of G that connects $T_u(e)$ and $T_v(e)$. We define for each edge e in T a recovery edge $rec(e)$. For the sake of the analysis, and to slightly simplify the routing phase, assume that the edges of the graph are sorted in some order $\langle e_1, \ldots, e_m \rangle$, and for every edge e, $rec(e)$ is chosen to be a recovery edge e_i of e such that i is minimal. We say that $e_i < e_j$ when $i < j$.

In order to handle two failures, we need to store additional information (i.e., additional recovery edges) in the routing tables of vertices in T. We show that the total number of recovery edges needed is within a constant factor from the size of T.

Consider the recovery edge $rec(e) = (u', v')$ of the edge e. The edge $rec(e)$ connects the subtrees $T_u(e)$ and $T_v(e)$ where $u' \in T_u(e)$ and $v' \in T_v(e)$. Denote by $P(u, u')$ (respectively, $P(v, v')$) the path connecting u and u' (respectively, v and v') in the tree $T_u(e)$ (respectively, $T_v(e)$), and denote the entire alternative path for $e = (u, v)$ by $P(e) = P(u, u') \cdot (u', v') \cdot P(v', v)$.

Throughout this section, assume the two failed edges are $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$. Clearly, if both e_1 and e_2 are not in T then we can just route on T. Hence, we only have to consider the case when T contains the failed edges.

We first consider the case that only one of the failed edges is in T. Assume, w.l.o.g., that $e_1 \in T$ and that $e_2 \notin T$. Notice that $T \cup \{rec(e_1)\} \setminus \{e_1, e_2\}$ is composed of two connected components only when $rec(e_1) = e_2$. To overcome this it suffices to store for each edge $e \in T$ an additional recovery edge. Then, in the scenario described above, where $rec(e_1) = e_2$ and $T \cup \{rec(e_1)\} \setminus \{e_1, e_2\}$ is not connected, we simply use the additional recovery edge of e_1 and we are guaranteed not to encounter additional faulty edges along the rest of the route. Note that if there is only one edge that can serve as a recovery edge for e_1 and this edge is faulty, then it is not possible to route from s to t on $G_i|_T \setminus \{e_1, e_2\}$. To summarize this case, for each edge e we store two recovery edges (if exist).

We now consider the case that both $e_1, e_2 \in T$. Let $rec(e_1) = (u'_1, v'_1)$ and $rec(e_2) = (u'_2, v'_2)$. If the edge e_2 is not on the alternative path $P(e_1) = P(u_1, u'_1) \cdot (u'_1, v'_1) \cdot P(v'_1, v_1)$ of e_1 and s and t are connected in $G_i|_T \setminus \{e_1, e_2\}$ then $rec(e_1)$ and $rec(e_2)$ suffice to route from s to t. The reason is that it is always possible to bypass e_1 using its alternative path $P(e_1)$. Therefore, the routing from s to t never gets stuck when reaching e_1. When the edge e_2 is encountered on the routing path it is bypassed using $P(e_2) = P(u_2, u'_2) \cdot (u'_2, v'_2) \cdot P(v'_2, v_2)$. If the edge e_1 is on $P(e_2)$, it is

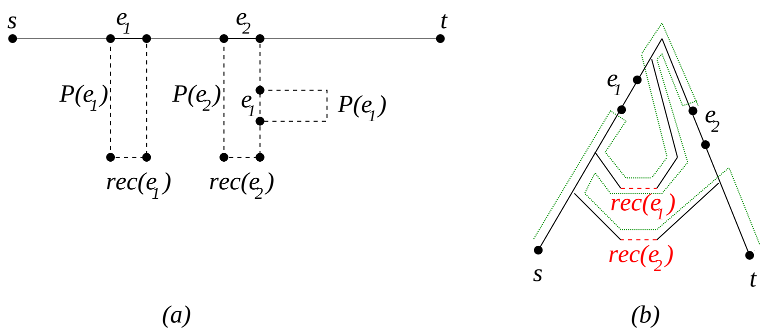

Figure 4: (a) A schematic description of an $s-t$ route where the failed edge e_1 is encountered twice. (b) The resulting route on the tree. Note that the alternate path $P(e_1)$ does not always have to be followed blindly; rather, it can be "shortcut" whenever the necessary information is readily available.

bypassed (again) using $P(e_1)$, which does not contain e_2. This situation is depicted in Figure 4. The situation that e_1 is not on $P(e_2)$ is symmetric. Therefore, it is only left to consider the situation in which both e_1 is in $P(e_2)$ and e_2 is in $P(e_1)$.

This implies that $rec(e_1) = rec(e_2)$. To see this, notice that since $P(e_2)$ contains e_1 the recovery edge $rec(e_2)$ is also a recovery edge for e_1, and similarly $rec(e_1)$ is also a recovery edge for e_2. Since we have chosen the recovery edges to be minimal with respect to a given ordering, it must be the case that $rec(e_1) = rec(e_2)$. Now since e_1 is in $P(e_2)$, e_2 is in $P(e_1)$ and $rec(e_1) = rec(e_2)$ it must be that $P(e_1) \cup \{e_1\} = P(e_2) \cup \{e_2\}$. To deal with this case, we store for e_1 (and similarly, for each edge $e \in T$) two additional recovery edges $rec_{u_1}(e_1)$ and $rec_{v_1}(e_1)$. The purpose of $rec_{u_1}(e_1)$ ($rec_{v_1}(e_1)$) is to handle an edge fault on $P(u_1, u'_1)$ ($P(v_1, v'_1)$). We choose $rec_{u_1}(e_1)$ such that it will allow to bypass as many edges on $P(u_1, u'_1)$ as possible. More specifically, consider the path from u_1 to any other recovery edge that differs from $rec(e_1)$. This path has some common prefix with $P(u_1, u'_1)$ (which is possibly empty). For $rec_{u_1}(e_1)$, we choose the recovery edge of e_1 that minimizes the length of this common prefix, that is, if $rec_{u_1}(e_1) = (\hat{u}, \hat{v})$ then the common prefix of $P(u_1, u'_1)$ and $P(u_1, \hat{u})$ is the minimal possible. The recovery edge $rec_{v_1}(e_1)$ is defined analogously with respect to $P(v_1, v'_1)$.

Next, we show how to route based on this information. Assume the failure of the set of edges $\{e_1, e_2\}$, and consider the task of routing from s to t. Let $T \in \mathbf{TC}_i$, for $i \in \{1, \ldots, \lceil \log nW \rceil\}$, be the tree on which s tries to route its message.

The simplest case is when the message reaches t and no forbidden edges were encountered on the route. So now consider the case where the message encounters the edge $e_1 = (u_1, v_1)$ on its route (the case where e_2 is encountered first is clearly symmetric).

We now describe how it is possible to route from s to t when e_1 is in $P(e_2)$, e_2 is in $P(e_1)$ and $rec(e_1) = rec(e_2) = e' = (u', v')$ using the additional information. The message first encounters the edge $e_1 = (u_1, v_1)$ on its route. If the recovery edge $rec(e_1)$ does not exist than t cannot be reached using T. If $rec(e_1) = (u', v')$ exists then the message is routed towards u' on $P(u_1, u')$. It uses (u', v') and continues to route from v' toward t on $P(v', v_1)$. Notice that at some stage along the path $P(u_1, u') \cdot (u', v') \cdot P(v', v_1)$, the edge e_2 is encountered. Since $rec(e_1) = rec(e_2) = e'$ it is not possible to bypass e_2 using $rec(e_2)$. There are two possible cases to consider here. The first is when the edge e_2 is on the path $P(u_1, u')$ and the second is when the edge e_2 is on the path $P(v_1, v')$. Notice that it is possible to distinguish between the two cases when e_2 is encountered simply by checking whether the edge $rec(e_1)$ was already traversed.

Consider the first case, where e_2 is on $P(u_1, u')$ and assume that v_2 is the endpoint of e_2 that is connected to u_1. There are three subtrees in $T \setminus \{e_1, e_2\}$. Let T_1 be the subtree containing u_1 and v_2. Let T_2 be the subtree containing u_2 and u' and let T_3 be the subtree containing v' and v_1. Note that if $t \in T$, then it must be that $t \in T_3$, as the routing scheme on T tries to send the message from s to t using $e_1 = (u_1, v_1)$ which implies that t is not in the subtree $T_1 \cup T_2 \cup \{e_2\}$ that contains u_1. Moreover, as we assume that we first encounter e_1 on the path from s to t in T, it holds that s is in T_1.

We first try to use the edge $rec_{u_1}(e_1)$. Recall that this edge was chosen such that the path leading to it from u_1 has the minimal possible common prefix with $P(u_1, u')$. Therefore, if there is a recovery edge $\tilde{r} = (\tilde{u}, \tilde{v})$ with endpoint $\tilde{u}$ in T_1 and $\tilde{v}$ in T_3, then clearly $rec_{u_1}(e_1) = (u'', v'')$ must be such an edge. To see this, assume towards contradiction, that the path $P(u_1, u'')$ contains the edge e_2. Note that the path $P(u_1, \tilde{u})$ contains fewer edges in common with $P(u_1, u')$ than the path $P(u_1, u'')$, in contradiction to the minimality of $P(u_1, u'')$. Therefore, if the subtree T_1 contains an edge leading to T_3, using the edge $rec_{u_1}(e_1)$ we can reach to the subtree T_3 containing t. See Figure 5(a) for illustration.

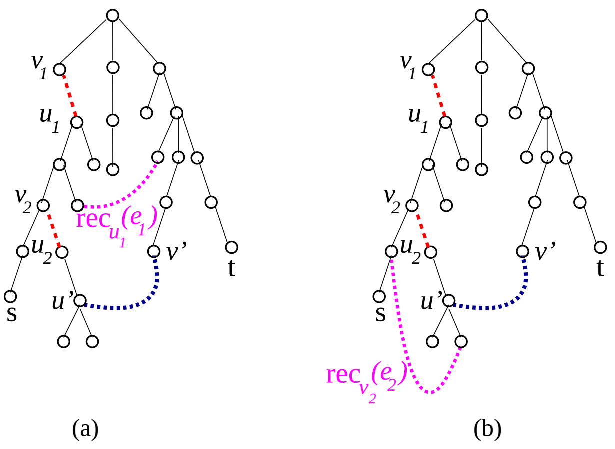

Figure 5: (a) The edge e_2 is on the path $P(u_1, u')$ and the edge $rec_{u_1}(e_1)$ is used. (b) The edge e_2 is on the path $P(u_1, u')$ and the edge $rec_{v_2}(e_2)$ is used.

The more involved subcase is when for every recovery edge $\tilde{r} = (\tilde{u}, \tilde{v})$ of e_1, the path $P(u_1, \tilde{u})$ contains the edge e_2, or in other words, there is no edge connecting the subtree T_1 with the subtree T_3 (except the faulty edge e_1). In this case, our only "chance" to reach t on the tree T is by passing through the trees T_1, T_2 and finally T_3 in this order. Notice that to connect between T_2 and T_3 we can use the edge $rec(e_1)$. Using ideas similar to those presented above, we show that it is possible to reach T_2 from T_1, and thus the additional information that we have saved will allow us to reach t.

Consider the case that there is a path between s and t in $G|_T \setminus \{e_1, e_2\}$. In this case, as s is in T_1 and there are no edges in $G|_T$ between T_1 and T_3 it must be the case that there is an edge between T_1 and T_2. Let $\tilde{r} = (\tilde{u}, \tilde{v})$ be such an edge. Now, by the definition of $rec_{v_2}(e_2)$ we may conclude that $rec_{v_2}(e_2)$ also connects between T_1 and T_2 (otherwise we would obtain a contradiction to the minimality condition in the definition of $rec_{v_2}(e_2)$). Therefore, we can use $rec_{v_2}(e_2)$ to reach the subtree T_2 and then use $rec(e_2) = rec(e_1) = e'$ to reach the desired subtree T_3 that contains t (see Figure 5(b)).

Finally, the analysis of the second case in which the edge e_2 is on the path $P(v_1, v')$ is similar to the analysis of the first case. Recall that in this case we assume that $rec(e_1) = rec(e_2) = e' = (u', v')$ and that e_2 is on the path $P(v_1, v')$. We also assume that we encounter e_2 while trying to route from v' to t The subgraph $T \setminus \{e_1, e_2\}$ contains three connected subtrees. Let T_1 be the subtree containing u_1, T_2 be the subtree containing v_2 and T_3 be the subtree containing both v_1 and u_2. Here, we assume that v_2 is the endpoint of e_2 closest to v'. As in previous case, we first try to use the edge $rec_{v_1}(e_1)$, whose goal is to bypass as many edges as possible in $P(v_1, v')$. Using the same arguments as before, if the subtree T_1 contains an edge leading to T_3, then $rec_{v_1}(e_1)$ must be such an edge and we can reach the subtree containing t (see Figure 6(a)).

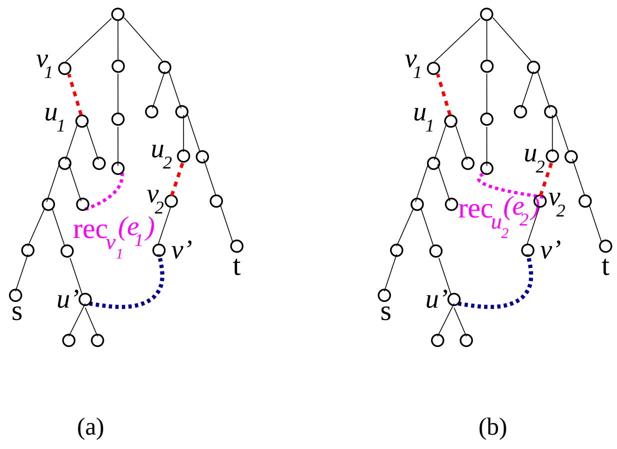

Figure 6: (a) The edge e_2 is on the path $P(v_1, v')$ and the edge $rec_{v_1}(e_1)$ is used. (b) The edge e_2 is on the path $P(v_1, v')$ and the edge $rec_{u_2}(e_2)$ is used.

So now consider the more involved subcase where for every recovery edge $\tilde{r} = (\tilde{u}, \tilde{v})$ of e_1, the path $P(v_1, \tilde{v})$ contains the edge e_2, or in other words, there is no edge connecting the subtree T_1 with the subtree T_3. Again, as before, our only "chance" to reach t is by passing through the trees T_1, T_2, and T_3 in that order, where we use the edge $rec(e_1)$ to get from T_1 to T_2.
Consider the case that there is a path between s and t in $G|_T \setminus \{e_1, e_2\}$. In this case, as s is in T_1 and there are no edges in $G|_T$ between T_1 and T_3 it must be the case that there is an edge between T_2 and T_3. Let $\tilde{r} = (\tilde{u}, \tilde{v})$ be such an edge. Now, by the definition of $rec_{u_2}(e_2)$ we may conclude that $rec_{u_2}(e_2)$ also connects between T_2 and T_3 (otherwise we would obtain a contradiction to the minimality condition in the definition of $rec_{u_2}(e_2)$). Therefore, we can use $rec(e_2) = rec(e_1) = e'$ to reach T_2 and $rec_{u_2}(e_2)$ to reach the desired subtree T_3 that contains t. (see Figure 6(b)).
We now turn to analyze the total stretch obtained by our routing scheme.

Lemma 3.2 *The resulting routing scheme has maximum stretch $O(k)$.*

Proof: Let $T \in \mathbf{TC}_i$, for $i \in \{1, \ldots, \lceil \log nW \rceil\}$, be the tree on which s tries to route its message. The depth of T is at most $2(k-1)2^i$, therefore any path between two nodes on the tree T is of length at most $4(k-1)2^i$ (twice the depth).
Our routing scheme first tries to route from s to t (this path is of length at most $4(k-1)2^i$). The route changes whenever the message encounters some unexpected "deviation event". For

instance, such an event occurs when we encounter some faulty edge $e_1 = (u_1, v_1)$. In this case we try to route from u_1 to $rec(e_1)$, and again some deviation event may occur (in this case the deviation events that may occur are either encountering another faulty edge or reaching $rec(e_1)$). Each such event changes the path on which we are currently route on the tree T. It's not hard to verify (using a detailed case analysis) that the number of such events in our routing scheme is bounded by a small constant (details omitted). Thus, the total length of the route followed in iteration i is at most $c \cdot k \cdot 2^i$ for some constant c.

In addition, our routing scheme may involve at most $\log(nW)$ iterations. Consider the first iteration i where the routing succeeds. The total length of the path is at most

$$\sum_{j=1}^{i} c \cdot k \cdot 2^j \ = \ O(k \cdot 2^i) \ .$$

The lemma follows as the exact distance from s to t in $G \setminus \{e_1, e_2\}$ is at least 2^{i-1}. □

All in all, for each edge $e = (u, v) \in T$, both endpoints u and v store three additional edges, $rec(e)$, $rec_u(e)$ and $rec_v(e)$. Theorem 1.2 follows.

References

[1] Y. Afek, H. Attiya, A. Fekete, M.J. Fischer, N. Lynch, Y. Mansour, D. Wang, and L.D. Zuck. Reliable communication over unreliable channels. *J. ACM*, 41:1267–1297, 1994.

[2] Y. Afek, B. Awerbuch, E. Gafni, Y. Mansour, A. Rosen, and N. Shavit. Slide-the key to polynomial end-to-end communication. *J. Algorithms*, 22:158–186, 1997.

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

[4] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *J. Algorithms*, pages 307–341, 1990.

[5] B. Awerbuch and S. Even. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing (PODC)*, pages 278–281, 1984.

[6] B. Awerbuch, S. Kutten, and D. Peleg. On buffer-economical store-and-forward deadlock prevention. In *Proc. INFOCOM*, pages 410–414, 1991.

[7] B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, October 1990.

[8] A. Bagchi and S.L. Hakimi. Information dissemination in distributed systems with faulty units. *IEEE Trans. Comp.*, 43:698–710, 1994.

[9] F. Bao, Y. Igarashi, and K. Katano. Broadcasting in hypercubes with randomly distributed byzantine faults. In *Proc. WDAG'95*, pages 215–229, 1995.

[10] S. Baswana and T. Kavitha. Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 591–602, 2006.

[11] S. Baswana and S. Sen. A simple linear time algorithm for computing sparse spanners in weighted graphs. In *Proc. 30th Int. Colloq. on Automata, Languages and Programming*, pages 384–396. SV, 2003.

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

[13] P. Berman, K. Diks, and A. Pelc. Reliable broadcasting in logarithmic time with byzantine link failures. *J. Algorithms*, 22:199–211, 1997.

[14] A. Bernstein and D. Karger. Improved distance sensitivity oracles via random sampling. In *Proc. 19th ACM-SIAM Symp. on Discrete algorithms (SODA)*, pages 34–43, 2008.

[15] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proc 41st ACM Symp. on Theory of computing (STOC)*, pages 101–110, 2009.

[16] D. Blough and A. Pelc. Optimal communication in networks with randomly distributed byzantine faults. *Networks*, 23:691–701, 1993.

[17] S. Chechik, M. Langberg, D. Peleg, and L. Roditty. Fault-tolerant spanners for general graphs. *SIAM Journal on Computing*, 37(7):3403–3423, 2010.

[18] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 648–658, 1993.

[19] B. Courcelle and A. Twigg. Compact forbidden-set routing. In *Proc. STACS*, pages 37–48, 2007.

[20] L.J. Cowen. Compact routing with minimum stretch. *J. Algorithms*, 38:170–183, 2001.

[21] C. Demetrescu and G.F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *Proc. 15th SODA*, pages 362–371, 2004.

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

[23] K. Diks and A. Pelc. Almost safe gossiping in bounded degree networks. *SIAM J. on Discrete Mathematics*, 5:338–344, 1992.

[24] D. Dolev. The byzantine generals strike again. *J. Algorithms*, 3:14–30, 1982.

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

[26] R. Duan and S. Pettie. Dual-failure distance and connectivity oracles. In *Proc. SODA*, 2009.

[27] T. Eilam, C. Gavoille, and D. Peleg. Compact routing schemes with low stretch factor. *J. Algorithms*, 46:97–114, 2003.

[28] M. Elkin. Computing almost shortest paths. *ACM Trans. Algorithms*, 1(2):283–323, 2005.

[29] D. Eppstein, Z. Galil, G. F. Italiano, and N. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. *JACM: Journal of the ACM*, 44, 1997.

[30] A. D. Fekete, N. Lynch, Y. Mansour, and J. Spinelli. The impossibility of implementing reliable communication in the face of crashes. *J. ACM*, 40:1087–1107, 1993.

[31] P. Fraigniaud and C. Gavoille. Memory requirement for universal routing schemes. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 223–230, 1995.

[32] E. Gafni and Y. Afek. End-to-end communication in unreliable networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 131–148, 1988.

[33] C. Gavoille and M. Gengler. Space-efficiency for routing schemes of stretch factor three. *J. Parallel Distrib. Comput.*, 61:679–687, 2001.

[34] C. Gavoille and D. Peleg. Compact and localized distributed data structures. *Distributed Computing*, 16:111–120, 2003.

[35] O. Goldreich, A. Herzberg, and Y. Mansour. Source to destination communication in the presence of faults. In *Proc 7th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 85–101, 1989.

[36] A. Herzberg and S. Kutten. Fast isolation of arbitrary forwarding faults. In *Proc. 8th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 339–353, 1989.

[37] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

[38] T. Kavitha. Faster algorithms for all-pairs small stretch distances in weighted graphs. In *Proc. FSTTCS*, pages 328–339, 2007.

[39] N. Khanna and S. Baswana. Approximate shortest paths oracle handling single vertex failure. In *Proc. STACS*, 2010.

[40] L. Kučera. Broadcasting through a noisy one-dimensional network. Technical Report MPI-I-93-106, Max-Planck-Institut, Saarbruecken, Germany, 1993.

[41] H. Nagamochi and T. Ibaraki. Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

[42] M. Pătraşcu and M. Thorup. Planning for fast connectivity updates. In *Proc. 48th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 263–271, 2007.

[43] A. Pelc. Fault-tolerant broadcasting and gossiping in communication networks. *Networks*, 28:143–156, 1996.

[44] A. Pelc and D. Peleg. Feasibility and complexity of broadcasting with random transmission failures. *Theoretical Computer Science*, 370:279–292, 2007.

[45] D. Peleg. *Distributed computing: a locality-sensitive approach.* SIAM, Philadelphia, PA, 2000.

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

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

[48] L. Roditty, M. Thorup, and U. Zwick. Deterministic constructions of approximate distance oracles and spanners. In *Proc. 32nd ICALP*, pages 261–272, 2005.

[49] L. Roditty, M. Thorup, and U. Zwick. Roundtrip spanners and roundtrip routing in directed graphs. *ACM Trans. Algorithms*, 4(3):1–17, 2008.

[50] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proc. 36th STOC*, pages 184–191, 2004.

[51] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proc. 9th SWAT*, pages 384–396, 2004.

[52] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proc. 37th ACM Symp. on Theory of Computing (STOC)*, pages 112–119, 2005.

[53] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. 13th ACM Symp. on Parallel algorithms and architectures (SPAA)*, pages 1–10, 2001.

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

[55] R. Thurimella. *Techniques for the design of parallel graph algorithms.* PhD thesis, Univ. of Texas, Austin, 1989.

[56] A. Twigg. *Compact forbidden-set routing.* PhD thesis, Cambridge University, 2006.