f-Sensitivity Distance Oracles and Routing Schemes

Shiri Chechik[1], Michael Langberg[2], David Peleg[1,⋆], and Liam Roditty[3]

[1] Department of Computer Science, The Weizmann Institute, Rehovot, Israel
{shiri.chechik,david.peleg}@weizmann.ac.il
[2] Computer Science Division, Open University of Israel, Raanana, Israel
mikel@openu.ac.il
[3] Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel
liamr@macs.biu.ac.il

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 $G(V, E \setminus F)$.

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 $O(kn^{1+1/k} \log(nW) \log n)$.

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 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

⋆ Supported in part by a France-Israel cooperation grant ("Mutli-Computing" project) from the Israel Ministry of Science.

M. de Berg and U. Meyer (Eds.): ESA 2010, Part I, LNCS 6346, pp. 84–96, 2010.

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 $G(V, E \setminus F)$.

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. *Let $f, k \geq 1$ be integer parameters. Let $F \subset E$ be a set of forbidden edges, where $|F| \leq f$. For a pair of vertices $s, t \in V$ let $d = \mathbf{dist}(s, t, G \setminus F)$. 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 returns in time $O(|F| \log^2 n \cdot \log\log n \cdot \log\log d)$ a* distance estimate $\tilde{d}$ *satisfying $d \leq \tilde{d} \leq (8k-2)(f+1) \cdot d$.*

Theorem 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 [15], Demetrescu et al. showed that it is possible to preprocess a directed weighted graph in time $\tilde{O}(mn^2)$ 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. In two recent papers [8,9], Karger and Bernstein improved the preprocessing time for 1-sensitivity queries to $O(n^2\sqrt{m})$ and then $\tilde{O}(mn)$, with unchanged size and query time.

In [17], 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 [17] 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 [17] 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 compu-

tation of approximate "all pairs" distances [5,25,19,1,16], spanners [30,31,11,6], distance oracles [39,32,7] and compact routing schemes [38,31].

In the approximate setting, when no forbidden edges are considered, distance oracles were introduced by Thorup and Zwick [39]. 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 $2k - 1$. 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 [39], while its size remains comparable to that of [39].

Very recently, and independently of our work, Khanna and Baswana [26] 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 [26]. The case of distance oracles that support multiple vertex faults remains open. In [10] 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 [14], 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 [36]. In the restricted case of unweighted undirected graphs, Roditty and Zwick [34] 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 [37] 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 $O(\sqrt{n})$ [20] and the best amortized update time is $O(\log^2 n)$ [24]. Pătraşcu and Thorup [28] considered a restricted model in which all the deleted edges are first deleted in a batch, and queries are answered next. They present a linear size data structure constructible in polynomial time that supports queries in $O(f \log^2 n \log\log n)$ worst-case time after a batch of f deletions. Similarly, our result implies that one can preprocess an undirected graph into a data structure that supports a batch of f deletions followed by a distance query in $O(f \cdot \log^2 n \cdot \log\log n \cdot \log\log d)$ time.

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 [23,29]. Following a sequence of improvements [31,2,4,13,18], the best currently known tradeoffs are due to Thorup and Zwick [38], 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 [31,21,22,39].

Fault-tolerant label-based distance oracles and routing schemes for graphs of bounded clique-width are presented in [12]. 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.

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 [4] and further developed in [3,11,33] (see [29]). 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 [28]. However, this alone does not suffice, as the oracle of [28] 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 [28] 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. 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 we associate with each edge of T a constant number of backup edges that are chosen carefully to satisfy certain conditions. Then, via case analysis, we show that these backup edges allow successful routing in the presence of two edge faults (that are not known in advance). The properties required of the selected backup edges are dictated by the structure of our case analysis. Our current techniques do not seem to extend to the general case of routing in the presence of f faults for $f \geq 3$.

Due to space limitations, some of our proofs are omitted.

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.

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 [29,3,11]. The second ingredient is the *connectivity oracle* recently presented in [28]. Finally, the third is a simple construction of a sparse subgraph that preserves connectivity with up to f failures. 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 1 ([3,11,29]). *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 ofnodes s and t can

answer efficiently whether s and t are connected in $G \setminus F$. The properties of the connectivity oracle of [28] are summarized in the following proposition. We use a slight variation of that construction, discussed after the proposition.

Proposition 2 ([28]). *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 [28] 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 [28]. 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 **Conn_Or** is $O(m)$. This is necessary in [28] 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 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 [35,27] how to construct fault-tolerant connectivity preservers with the desired properties. A closely related problem is the k-edge witness problem studied in [40]. 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 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.

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 also show:

Lemma 1. *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 $H' = (V, E_{PR} \setminus F)$.*

Our construction: We now describe our construction of $\mathbf{Sens\_Or}(G, \omega, f, \mathcal{K})$, where $\mathcal{K} = (8k-2)(f+1)$ for integers k and f.

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

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 $G_i|_{T_i(s)}$ after the set F of forbidden edges is excluded from it. This is done by querying the connectivity oracle $\mathbf{Conn\_Or}_{T_i(s)}$ of $T_i(s)$ with (s, t, F). 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 ∞.

Theorem 1 now follows by the lemmas below.

Lemma 3 (Correctness). *The f-sensitivity distance query algorithm returns an estimate that is within a multiplicative factor of $\mathcal{K}$ from* $\mathbf{dist}(s,t,G \setminus F)$.

Lemma 4. *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 is the distance between s and t in $G \setminus F$.*

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 2. Let $s,t \in V$ and assume that a message is to be routed from s to t. 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 [38]. 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 [38] may assign a node t 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 $1 \leq i \leq \log(nW)$ (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.

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 $O(|T|)$.

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 $P(e_1) = P(u_1, u'_1) \cdot P(u'_1, v'_1) \cdot P(v'_1, v_1)$ 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 $P(e_2) = P(u_2, u'_2) \cdot P(u'_2, v'_2) \cdot P(v'_2, v_2)$. If the edge e_1 is on $P(e_2)$, it is bypassed (again) using $P(e_1)$, which does not contain e_2. This situation is depicted in Figure 1. 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 $P(e_1)$.

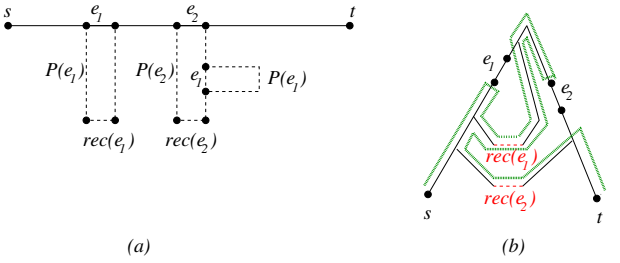

Fig. 1. (a) A schematic description of an $s-t$ route where the failed edge e_1 is encountered twice. (b) The resulting route on the tree. Note that the alternate path $P(e_1)$ does not always have to be followed blindly; rather, it can be "shortcut" whenever the necessary information is readily available.

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 $P(v_1, v_1')$.

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.

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. The details, as well as the analysis of the second case, in which the edge e_2 is on the path $P(v_1, v')$, are omitted.

Lemma 5. *The resulting routing scheme has maximum stretch $O(k)$.*

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 2 follows.

Acknowledgement. We are grateful to Seth Pettie for his helpful comments.

References

1. Aingworth, D., Chekuri, C., Indyk, P., Motwani, R.: Fast estimation of diameter and shortest paths (without matrix multiplication). SIAM J. Comput. 28(4), 1167–1181 (1999)
2. Awerbuch, B., Bar-Noy, A., Linial, N., Peleg, D.: Improved routing strategies with succinct tables. J. Algorithms, 307–341 (1990)
3. Awerbuch, B., Kutten, S., Peleg, D.: On buffer-economical store-and-forward deadlock prevention. In: INFOCOM, pp. 410–414 (1991)
4. Awerbuch, B., Peleg, D.: Sparse partitions. In: 31st FOCS, pp. 503–513 (1990)
5. Baswana, S., Kavitha, T.: Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In: FOCS, pp. 591–602 (2006)
6. Baswana, S., Sen, S.: A simple linear time algorithm for computing sparse spanners in weighted graphs. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 384–396. Springer, Heidelberg (2003)
7. Baswana, S., Sen, S.: Approximate distance oracles for unweighted graphs in expected $O(n^2)$ time. ACM Trans. Algorithms 2(4), 557–577 (2006)

8. Bernstein, A., Karger, D.: Improved distance sensitivity oracles via random sampling. In: 19th SODA, pp. 34–43 (2008)
9. Bernstein, A., Karger, D.: A nearly optimal oracle for avoiding failed vertices and edges. In: 41st STOC, pp. 101–110 (2009)
10. Chechik, S., Langberg, M., Peleg, D., Roditty, L.: Fault-tolerant spanners for general graphs. In: 41st STOC, pp. 435–444 (2009)
11. Cohen, E.: Fast algorithms for constructing t-spanners and paths with stretch t. In: FOCS, pp. 648–658 (1993)
12. Courcelle, B., Twigg, A.: Compact forbidden-set routing. In: STACS, pp. 37–48 (2007)
13. Cowen, L.J.: Compact routing with minimum stretch. J. Alg. 38, 170–183 (2001)
14. Demetrescu, C., Italiano, G.F.: Experimental analysis of dynamic all pairs shortest path algorithms. In: 15th SODA 2004, pp. 362–371 (2004)
15. Demetrescu, C., Thorup, M., Chowdhury, R., Ramachandran, V.: Oracles for distances avoiding a failed node or link. SIAM J. Comput. 37(5), 1299–1318 (2008)
16. Dor, D., Halperin, S., Zwick, U.: All-pairs almost shortest paths. SIAM J. Comput. 29(5), 1740–1759 (2000)
17. Duan, R., Pettie, S.: Dual-failure distance and connectivity oracles. In: SODA (2009)
18. Eilam, T., Gavoille, C., Peleg, D.: Compact routing schemes with low stretch factor. J. Algorithms 46, 97–114 (2003)
19. Elkin, M.: Computing almost shortest paths. ACM Trans. Alg. 1, 283–323 (2005)
20. Eppstein, D., Galil, Z., Italiano, G.F., Nissenzweig, N.: Sparsification – A technique for speeding up dynamic graph algorithms. J. ACM 44 (1997)
21. Fraigniaud, P., Gavoille, C.: Memory requirement for universal routing schemes. In: 14th PODC, pp. 223–230 (1995)
22. Gavoille, C., Gengler, M.: Space-efficiency for routing schemes of stretch factor three. J. Parallel Distrib. Comput. 61, 679–687 (2001)
23. Gavoille, C., Peleg, D.: Compact and localized distributed data structures. Distributed Computing 16, 111–120 (2003)
24. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J. ACM 48(4), 723–760 (2001)
25. Kavitha, T.: Faster algorithms for all-pairs small stretch distances in weighted graphs. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 328–339. Springer, Heidelberg (2007)
26. Khanna, N., Baswana, S.: Approximate shortest paths oracle handling single vertex failure. In: STACS (2010)
27. Nagamochi, H., Ibaraki, T.: Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph. Algorithmica 7, 583–596 (1992)
28. Pătraşcu, M., Thorup, M.: Planning for fast connectivity updates. In: 48th FOCS, pp. 263–271 (2007)
29. Peleg, D.: Distributed computing: a locality-sensitive approach. In: SIAM (2000)
30. Peleg, D., Ullman, J.D.: An optimal synchronizer for the hypercube. SIAM J. Computing 18(4), 740–747 (1989)
31. Peleg, D., Upfal, E.: A trade-off between space and efficiency for routing tables. J. ACM 36(3), 510–530 (1989)
32. Roditty, L., Thorup, M., Zwick, U.: Deterministic constructions of approximate distance oracles and spanners. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 261–272. Springer, Heidelberg (2005)

33. Roditty, L., Thorup, M., Zwick, U.: Roundtrip spanners and roundtrip routing in directed graphs. ACM Trans. Algorithms 4(3), 1–17 (2008)
34. Roditty, L., Zwick, U.: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In: 36th STOC, pp. 184–191 (2004)
35. Thurimella, R.: Techniques for the design of parallel graph algorithms. Ph.D. Thesis. Univ. Texas, Austin (1989)
36. Thorup, M.: Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 384–396. Springer, Heidelberg (2004)
37. Thorup, M.: Worst-case update times for fully-dynamic all-pairs shortest paths. In: 37th STOC, pp. 112–119 (2005)
38. Thorup, M., Zwick, U.: Compact routing schemes. In: SPAA, pp. 1–10 (2001)
39. Thorup, M., Zwick, U.: Approximate distance oracles. J. ACM 52, 1–24 (2005)
40. Twigg, A.: Compact forbidden-set routing. Ph.D. Thesis. Univ. Cambridge (2006)