Simple and practically efficient fault-tolerant 2-hop cover labelings

Feliciano Colella[1], Mattia D'Emidio[1], Guido Proietti[2,3]

[1] Gran Sasso Science Institute (GSSI), L'Aquila
[2] Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, Università degli Studi dell'Aquila.
[3] Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", Consiglio Nazionale delle Ricerche.
{mattia.demidio, feliciano.colella}@gssi.it, guido.proietti@univaq.it

Abstract A *labeling scheme* is a method to retrieve some global property of a graph by exploiting (in a distributed fashion) the information stored, in the form of (short) labels, at its vertices. A classic question in the area is that of labeling the vertices in order to infer the distance between any pair of them, or alternatively to retrieve a corresponding shortest path. On such purpose, among the various solutions, a *2-hop cover distance/path-reporting labeling scheme* is based on the idea of representing the shortest paths in a graph as the concatenation at an intermediate so-called *hub* vertex of the two shortest paths emanating from the corresponding end vertices. The difficult point here is to find a small-size set of hub vertices covering a shortest path for each pair of vertices in the graph, since in this way the corresponding labeling will be compact, but practical efficient solutions are known in the literature. In this paper, we present a simple and efficient way of enriching this successful scheme in order to make it *resilient* to the failure of a subset of at most $k \geq 1$ edges in the graph, by exploiting the notion of so-called *edge-independent* spanning trees. Depending on k, we provide different strategies and analyze the corresponding performances. In addition, to assess the practical effectiveness of our method, we provide an experimental evaluation, conducted for the case $k = 1$. Our data confirms the new method to be really competitive when compared with the performance of the benchmark non-fault-tolerant scheme, equipped with a point-of-failure rerouting policy.

1 Introduction

A *shortest-path query* asks the shortest path between two vertices in a graph. Without doubt, answering to this kind of queries is one of the most fundamental operations on graphs, as it has a wide range of applications, e.g. social networks analysis [20], route planning in road networks [12, 13], intelligent transport systems [5], routing in communication networks [9, 11]. Baseline methods, e.g. Breadth First Search or Dijkstra's algorithm, yield unsustainable *query times* to answer such requests in graphs arising from the mentioned applications, which tend to be huge.

To overcome this limit, tens of smarter approaches have been proposed in the last decade [2, 12] which adopt the common strategy of preprocessing the graph in advance to speed-up the query phase. Among them, the *2-hop cover distance/path-reporting labeling scheme* (2–HCL from now on) is based on the idea of representing the shortest paths of a graph as the concatenation at an intermediate so-called *hub* vertex of the two shortest paths emanating from the corresponding end vertices. Roughly speaking, the label at each vertex will contain the length and the next-hop vertex of a shortest path towards each hub vertex, and to retrieve a shortest path between two vertices it will suffice to join their labels and find the common hub that minimizes the sum of the two distances. The difficult point here is to find a small set of hub vertices covering a shortest path for each pair of vertices in the graph, since in this way the corresponding labeling will be compact. Indeed, finding a minimum-size set of hub vertices is known to be NP-hard [6].

Nevertheless, in the practice, 2–HCL is currently considered the best scheme since: i) it is general (it can be applied on all kinds of graphs, including directed weighted ones [10, 13]); ii) even in networks with tens of billions of vertices, it combines extremely low query times with affordable space overhead and reasonable preprocessing effort [1]; iii) it is suited to be used in distributed settings [21]; iv) several practical approximation algorithms are known [6] - among them, the recent *pruned landmark labeling* (PLL) [1] achieves considerably better scalability than other methods.

A further level of complexity, when dealing with this kind of approaches, is represented by the fact that, in general, real-world networks are prone to failures. In this case, in fact, if a failing edge is on a queried shortest path, then a distance query will return an underestimated values w.r.t. a true shortest path in the damaged graph, while a path-reporting query would actually return an unfeasible path!

In general, reprocessing the graph from scratch after each failure is not a reasonable recovery strategy, since it will unavoidably require a non-negligible amount of computational time [10]. Therefore, a desirable objective becomes that of devising a so-called *fault-tolerant* scheme, i.e. an approach that allows to answer to queries even in presence of a number of (transient) graph failure operations (e.g., edge or vertex removals). This is usually achieved by suitably *enriching* the underlying data structure and by accordingly modify the query strategy to consider such enrichment. However, if the paths to be reported are constrained to be *shortest* also in presence of failures, then this might result in storing an unpractical amount of additional information [17]. Hence, in this case, a reasonable compromise is that of relaxing the optimality constraint and devise more compact schemes that return *approximate* distances (shortest paths, resp.), i.e. distances (paths, resp.) that are guaranteed to be longer than the corresponding true distances (shortest paths, resp.) by at most a given *stretch factor*, for any possible kind of failure that has to be handled. Examples of this strategy in the literature are various [3, 8], where many structures have been

studied in the fault-tolerant approximate setting. However, to the best of our knowledge, no previous work has ever investigated 2–HCL in such a setting.

Our Contribution. In this paper, we move forward in this direction and present a simple and efficient way of enriching a 2–HCL in order to make it *resilient* to the failure of a subset of any k edges. In more details, given an n-vertex and m-edge graph G, we show that such an enrichment can be done by exploiting the notion of so-called *edge-independent* spanning trees. More precisely:

- for $k = 1, 2, 3$ and G at least $(k+1)$-edge connected, the enrichment takes $O(m+n)$, $O(n^2)$ and $O(n^3)$ additional time, respectively, and $O(n)$ additional space;
- for $k > 3$ and G at least $(2k+2)$-edge connected, the enrichment takes $O(k^2n^2)$ additional time and $O(kn)$ additional space.

Even though the above solutions have in principle replacement paths which can be linearly (in n and k) stretched (as compared to new shortest paths in the surviving graph), in practice the situation is much better. To assess that, we provide an experimental evaluation, conducted for the case $k = 1$, showing the quality of our approach in terms of stretch, space requirements and preprocessing/query time. Our data shows the new method to be really effective and competitive when compared with the performance of the benchmark non-fault-tolerant scheme, equipped with a point-of-failure rerouting policy. Notice that our results can be extended to weighted graphs as well, and that our solutions are able to answer to the most general path-reporting queries.

2 Background

In what follows, we provide the notation and the basic definitions that are used throughout the paper. Given an unweighted undirected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, we denote by $e = (u, v)$ an edge $e \in E$ connecting two vertices $u, v \in V$. Given an edge $e \in E$ of G, we denote by $G - e$ ($G - F$, resp.) the graph obtained from G by removing an edge e (a set of edges $F \subseteq E$, resp.). Moreover, we call π^G_{xy} a shortest path between two vertices $x, y \in V$ and d^G_{xy} (or $|\pi^G_{xy}|$) its length (a.k.a. the *distance* between x and y in G).

In addition, given a shortest path π^G_{xy} we denote by $p(x, \pi^G_{xy})$ ($p(y, \pi^G_{xy})$, resp.) the *predecessor* of x (y, resp.) within π^G_{xy}, i.e. the vertex v such that $(x, v) \in \pi^G_{xy}$ ($(v, y) \in \pi^G_{xy}$, resp.). Given two simple paths a and b, we denote by $p = a \oplus b$ the path p obtained by concatenating them.

Furthermore, we say that a graph $G = (V, E)$ is k edge connected if $|E| > k$ and there does not exist a set of edges, of cardinality at most $k-1$, whose removal disconnects the graph. Given a graph $G = (V, E)$ and a distinguished *root* vertex $r \in V$, then $\mathtt{IT} = \{T_1, T_2, \ldots, T_q\}$ is a collection of q *edge-independent* spanning trees of G if and only if, for each vertex $v \in V$, and for each $i \neq j$, $\pi^{T_i}_{rv}$ and $\pi^{T_j}_{rv}$ are pairwise edge-disjoint paths, i.e. they do not share any edge [15]. In what

follows, we give a slightly modified definition of 2–HCL, suited for answering to path-reporting queries, inspired by that of [6] for distance queries.

Definition 1 (2–HCL). *Let $G = (V, E)$ be an undirected graph. Let the* path label *$P(v)$ of a vertex $v \in V$ be a set of tuples of the form $\langle u, d^G_{uv}, p(v, \pi^G_{uv}))\rangle$ and let $P(G) = \{P(v)\}_{v \in V}$ (denoted as P when the graph is clear from the context) be a collection of such labels. Let a path query operation on $P(G)$ from a vertex s to a vertex t be defined as:*

$$\textsc{pQuery}(s, t, P) = \begin{cases} \langle h, d^G_{hs}, p(s, \pi^G_{hs})\rangle \mid h = \underset{v \in P(s) \cap P(t)}{\operatorname{argmin}} \{d^G_{vs} + d^G_{vt}\} & \text{if } P(s) \cap P(t) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

where, for the sake of brevity, we denote by $P(s) \cap P(t)$ the set of vertices $\{v | \langle v, d^G_{vs}, p(v, \pi^G_{vs})\rangle \in P(s) \wedge \langle v, d^G_{vt}, p(v, \pi^G_{vt})\rangle \in P(t)\}$. Then, $P(G)$ is a 2–HCL of G if and only if, for any pair of vertices s and t of G it satisfies the cover property [6], i.e., if s and t are connected in G and h is the first argument returned by $\textsc{pQuery}(s, t, P)$*, then $d^G_{st} = d^G_{hs} + d^G_{ht}$ and h is called a* hub *that* covers *s and t, otherwise* $\textsc{pQuery}(s, t, P) = \emptyset$.

Given the above, it is clear that a 2–HCL $P(G)$ of a graph G can be used also to retrieve an entire shortest path π^G_{st} (i.e. the corresponding set of vertices/edges) for any given pair of vertices s, t of G by the procedure shown in Algorithm 1. The routine essentially builds incrementally two shortest paths, one from s toward a hub h and the other from t toward the same hub h, and eventually combines them into a path from s to t which is exactly the shortest between s and t, by the cover property. The two paths are computed by the repeated application of the above path query, i.e. by concatenating edges connecting vertices to the corresponding predecessors. For the sake of clarity, we represent the shortest path as a doubly linked list of edges.

Algorithm 1: Computation of a shortest path via $P(G)$.

Input : $P(G)$, a pair of vertices x and y of G.
Output: A shortest path $p = \pi^G_{xy}$ between x and y.

```
1  p ← ∅; p_x ← ∅; p_y ← ∅; c_1 ← x; c_2 ← y;
2  if c_1 = c_2 then return p;
3  l_1 ← pQuery(c_1, c_2, P);
4  l_2 ← pQuery(c_2, c_1, P);
5  h ← l_1.first                          /* h equals l_2.first by construction */
6  while c_1 ≠ l_1.first ∨ c_2 ≠ l_2.first do
7  |  if c_1 ≠ l_1.first then
8  |  |  p_x.pushFront((c_1, l_1.third));
9  |  |  c_1 ← l_1.third;
10 |  |  l_1 ← pQuery(c_1, h, P);
11 |  if c_2 ≠ v then
12 |  |  p_y.pushBack((c_2, l_2.third));
13 |  |  c_2 ← l_2.third;
14 |  |  l_2 ←← pQuery(c_2, h, P);
15 p = p_x ⊕ p_y;
16 return p
```

Problem Statement Let $G = (V, E)$ be an unweighted and undirected graph. We aim at computing what we call a *k-edge-fault-tolerant path-reporting labeling scheme* (k-EFTPL, in short), i.e. a labeling $P(G, k)$ that, for any pair of vertices s, t of G, can be used to retrieve: i) a shortest path π_{st}^{G} between s and t in G; ii) a *backup path* γ_{st}^{G-F} between s and t, i.e. a simple path (if any) connecting s and t in $G - F$ for any $F \subseteq E$ such that $|F| \leq k$.

The stretch factor of a k-EFTPL $P(G, k)$ is defined as the maximum ratio, for any pair of vertices s, t and for any $F \subseteq E$ such that $|F| \leq k$, between the length of a backup path and that of the corresponding shortest path in $G - F$, i.e.: $\alpha(P(G,k)) = \max_{s,t \in V, \forall F \subseteq E : |F| \leq k} \frac{|\gamma_{st}^{G-F}|}{|\pi_{st}^{G-F}|}$. Note that, whenever π_{st}^{G-F} does not exist (i.e. s and t are disconnected in $G - F$), we assume $\frac{|\gamma_{st}^{G-F}|}{|\pi_{st}^{G-F}|}$ to be 1. A k-EFTPL $P(G, k)$ such that $\alpha(P(G, k)) = 1$ is called *exact* while is called *approximate* otherwise.

3 A linear-stretch approximate *k*-EFTPL

In this section, we show how to build an approximate k-EFTPL $P(G, k)$ for a given input graph G. Our approach essentially consists in a suitable enrichment of a 2–HCL $P(G)$, based on the use of edge-independent trees, as follows. Given G, we start by computing $P(G)$ via a standard method, e.g. PLL [1] and, to be able to "resist" to the failure of k edges, we compute $k+1$ edge-independent trees $T_1, T_2, \ldots T_{k+1}$ of G and root them all at a same distinguished vertex, say $r \in V$. Note that, such trees guarantess that, should any k edges $e_1, e_2, \ldots, e_k$ of the graph fail, for any vertex $v \in V$ there will exist (at least) one value $i \in [1, k+1]$ such that v and r are connected, in T_i, by a simple path not containing any of the $e_1, e_2, \ldots, e_k$ edges. This can be trivially proved by contradiction, by elaborating on the well-known results from graph theory, summarized in the following.

Theorem 1 ([16]). *Let IT be a graph obtained by merging a collection of $q+1$ edge-independent spanning trees $\{T_i\}_{i=1,2,\ldots,q+1}$ of a graph G. Then,* IT *is $(q+1)$-edge connected.*

Hence, we use such rooted trees to build a set of what we call *tree labels* that are assigned to the vertices in order to allow the retrieval of a backup path in presence of k edge failures. Intuitively, such labels store the predecessor of a vertex within a shortest path toward the root r, for each considered tree.

Depending on the value of k, we make use of different approaches to build the $k + 1$ edge-independent trees. Clearly, a necessary condition to guarantee that any pair of vertices remains connected even in presence of k edge failures is that G is at least $(k + 1)$-edge connected. Then, if $k \in \{1, 2, 3\}$ and G is $(k + 1)$-edge connected, we can compute $k + 1$ edge-independent trees in polynomial time, with a time complexity of $O(m + n)$, $O(n^2)$ and $O(n^3)$, respectively [4, 7, 15].

On the other hand, if $k > 3$ and G is h-edge connected, with $k + 1 \leq h \leq 2k + 1$, then to the best of our knowledge it is not possible to build $k + 1$ edge-independent trees in polynomial time. However, if G is at least $(2k + 2)$-edge

connected, then we can build $k+1$ *edge-disjoint* spanning trees of G, which are clearly also edge-independent, by using the approach proposed in [18], running in $O(k^2n^2)$ time.

From now on, we will suppose to have at our disposal $k+1$ edge-independent trees of G, built as aforementioned. Hence, for each vertex, there will always be a tree label containing a viable predecessor to be used to reach r, even in presence of k arbitrary edge failures in G. More in details, given a vertex $v \in V$ and an independent tree T_i of G (rooted at a vertex $r \in V$), we define a *tree label entry* to be a pair $\langle r, p(v, \pi_{vr}^{T_i})\rangle$. Such pair essentially encodes for v a path to follow toward the root r of the i-th tree in the form of a predecessor within the tree. Tree label entries are stored within what we call a *tree label* $T(v,k)$,[4] for any $v \in V$. Hence, any vertex is assigned two labels, namely $P(v)$ and $T(v,k)$, to be used in combination in order to be tolerant to the failure of k edges. We call the set $\{T(v,k)\}_{v\in V}$ a *k-tree labeling* of G (denoted as $T(G,k)$ or T when clear from the context) while we define $P(G,k)$ to be the union of $P(G)$ and $T(G,k)$. Moreover, we call the chosen r the *root* of $P(G,k)$.

The routine for building the k-tree labeling of a graph G, and therefore $P(G,k)$, is summarized in Algorithm 2. We represent $T(v,k)$ again as a doubly linked list, since it will be convenient for defining a suited query algorithm. Now, we can easily bound the computational complexity of Algorithm 2 as follows.

Algorithm 2: Building a k-EFTPL $P(G,k)$ of a graph $G=(V,E)$

Input : Graph $G=(V,E)$, parameter $k \geq 1$, a 2–HCL $P(G)$ of G
Output: A k-EFTPL $P(G,k)$ of G

1 Compute $k+1$ edge-independent trees $T_1, T_2, \ldots T_{k+1}$;
2 Choose a distinguished vertex $r \in V$, root all k trees in r;
3 $T(G,k) \leftarrow \emptyset$;
4 **foreach** $v \in V$ **do**
5 | **foreach** $i = 1, 2, \ldots, k+1$ **do** $T(v,k).push\_back(\langle r, p(v, \pi_{vr}^{T_i})\rangle)$;
6 | $T(G,k) \leftarrow T(v,k)$;
7 $P(G,k) = P(G) \cup T(G,k)$;
8 **return** $P(G,k)$

Lemma 1. *Algorithm 2 takes $O(\Delta + k|V|)$ time, where Δ is the worst-case time for computing $k+1$ edge-independent trees.*

Proof. Clearly, performing line 1 takes $O(\Delta)$ while the execution of lines 2-3 takes constant time. Moreover, the outer loop of line 4 is executed exactly $|V|$ times while the inner loop of line 5 is performed exactly $k+1$ times. Since the operation of line 5 can be done in $O(1)$ time,[5] the claim follows. □

Lemma 2. *Algorithm 2 builds a k-EFTPL $P(G,k)$ whose overall size, in terms of label entries, is $O(|P(G)| + k|V|)$.*

Proof. To prove the statement, it is enough to bound the size of $T(G,k)$, since the size of $P(G,k)$ is trivially upper bounded by the sum of the sizes of $P(G)$ and $T(G,k)$. At line 3, $T(G,k)$ is empty, and exactly one entry per value of i is added to the tree label $T(v,k)$ of each vertex $v \in V$ at line 5. Since i ranges from 1 to k, the claim follows. □

[4] The k parameter stands for the ability of resisting to k edge failures by $k+1$ trees.
[5] By storing each T_i as an array of predecessors.

Given the construction of Algorithm 2, we now provide a query algorithm that can be used for either retrieving a true shortest path, when the graph is not subject to failures, or a backup path otherwise. The procedure, summarized in Algorithm 3, roughly works as follows. Given $P(G,k)$ and a pair of vertices s and t, the routine starts by trying to retrieve the true shortest path, by relying on $P(G)$ and on the corresponding pQuery strategy. As in the previous case, the shortest path is built incrementally and by combining two separate parts. If no failures are currently present in the network, then the algorithm essentially coincides with Algorithm 1 and returns the shortest path. Otherwise, if a number $k' < k+1$ of failures are occurring on G, then two cases can occur: either the shortest path from s to t includes at least one of such failures or not. To determine that, the algorithm needs essentially to check, at every step of the construction, whether the edge leading to the predecessor on the original shortest path has failed or not. In the negative case, clearly, the algorithm proceeds as nothing happened, i.e. as Algorithm 1, by repeating the application of pQuery. In the affirmative case, instead, the algorithm starts looking for backup paths by using available tree label entries, which essentially encode edges connecting to predecessors in the independent trees (see Lines 4-9 of Algorithm 4). Since a number of failures can affect different trees, tree label entries are scanned sequentially until one non-failed edge if found. Again two sub-paths are built and then, eventually, combined.

Algorithm 3: Query algorithm for a k-EFTPL $P(G,k)$.

```
   Input  : P(G,k), two vertices x and y.
   Output: A simple path p between x and y.
 1 p ← ∅; p_x ← ∅; p_y ← ∅; c_1 ← x; c_2 ← y;
 2 l_0 ← the 0-th tree label of T(z,k);    /* Any tree label entry, first field is always r */
 3 r ← l_0.first;
 4 if x = y then return p;
 5 l_1 ← pQuery(c_1, c_2, P);
 6 h ← l_1.first                                  /* h equals l_2.first by construction */
 7 p_x ← NavigateFrom(x, h, r); p_y ← NavigateFrom(y, h, r);
 8 if p_x.tail() ≠ p_y.tail() then
 9 |   if p_x.tail() ≠ h then ForceToRoot(p_x, r);
10 |   else ForceToRoot(p_y, r);
11 else p ← p_x ⊕ reverse(p_y);
12 for e ∈ p do
13 |   if |e| > 1 then  p ← p \ e                  /* Removes any duplicates */ ;
14 return p;
```

Note that, in this case, the two sub-paths might be not connected since one of the two searches might end up in the original hub, while the other in the root, and viceversa. This happens when one of the two sub-paths reaches the original hub while the other does not. In this case, Algorithm 5 takes care of finding the missing part. In general, we are able to prove the following.

Lemma 3. *Algorithm 3 always computes a backup path connecting x to y, for any pair $x, y \in V$.*

Due to space contraints, we postpone the proof of Lemma to a full version of the paper.

Theorem 2. *Algorithm 3 builds a k-EFTPL $P(G,k)$ with $\alpha(P(G,k)) \in O(kn)$.*

Algorithm 4: Procedure NAVIGATEFROM(z, h, r)

```
   Input  : P(G,k), a vertex z, its hub h, the root of the trees r.
   Output: A simple path p_z connecting z to either h or r.
 1 p_z ← ∅; FAILED ← false; i ← 0;                    /* i-th edge-independent tree in use */
 2 l_z ← PQUERY(z, h, P);
 3 while z ≠ h ∧ z ≠ r do
 4 |   if FAILED = true then      /* failure previously encountered, keep going toward r */
 5 |   |   Let l_i be the i-th tree label of T(z,k);
 6 |   |   if (z, l_i.second) ∈ E then
 7 |   |   |   p_z.pushFront((z, l_i.second));
 8 |   |   |   z ← l_i.second;
 9 |   |   else i ← i + 1;                                          /* change tree */
10 |   else
11 |   |   if (z, l_z.third) ∉ E then FAILED ← true;          /* failure encountered */
12 |   |   else                                          /* keep navigating toward h */
13 |   |   |   p_z.pushFront((z, l_z.third));
14 |   |   |   z ← l_z.third;
15 |   |   |   l_z ← PQUERY(z, h, P);
16 return p_z;
```

Proof. The length of any path found by Algorithm 3 from any node v to the root r is always upper bounded by $\text{DIAM}_{max} = \max_{i=1,\ldots,k+1} 2 \cdot \text{DIAM}(T_i)$, where $\text{DIAM}(T_i)$ is the diameter of the i-th edge-independent tree T_i. Since we could find a failed edge up to k times (one for each tree T_i), we could have to pay $\text{DIAM}(T_i)$ each time. Therefore we could have $k \cdot \text{DIAM}(T_i)$, hence, the claim follows. □

Algorithm 5: Procedure FORCETOROOT(p, r).

```
   Input  : P(G,k), a path p, a vertex r.
 1 i ← 0; z ← p.tail();
 2 while z ≠ r do
 3 |   Let l_i be the i-th tree label of T(z,k);
 4 |   if (z, l_i.second) ∈ E then
 5 |   |   p.pushFront((z, l_i.second));
 6 |   |   z ← l_i.second;
 7 |   else i ← i + 1;                                              /* change tree */
```

4 Experimental evaluation

In this section we provide an experimental evaluation of our approach. In particular, we implemented, in C++: i) the PLL approach of [1] for building 2–HCL; ii) Algorithm 2 for building an approximate k-EFTPL, for $k = 1$. Note that, details on how to build a 1-EFTPL will be given in an extended version of the paper due to space constraints. We embedded all our code within *NetworKit*[6], a well-known open-source framework for large-scale network analysis. Then, to assess the performance of our method, we established the following experimental process.

First, inspired by other papers on the subject (e.g. [10, 17]), we selected a meaningful set of real-world networks of various types (including road graphs,

[6] see networkit.iti.kit.edu

peer to peer networks, and AS communication networks[7]) and treated them as undirected, unweighted graphs. Details about inputs are reported in Table 1 where we show, for each network, the corresponding type, and the number of vertices and edges of the largest 2-edge connected component (second, third and fourth columns, resp.). Then, for each network, we ran both PLL and Algorithm 2

Network	**Type**	$\|V\|$	$\|E\|$	**time** (seconds)		**space per vertex** (bytes)	
				PLL	KHL	$P(G)$	$T(G,1)$
BARABASI (BAR)	Synthetic	365 488	734 347	68 500	6.41	5 198	18
BRIGHTKITE (BRI)	Social	33 187	188 577	5 680	1.75	2 326	18
CA-GRQC (CAG)	Collaboration	2 651	10 480	499	0.03	1 271	18
CA-HEPTH (CAH)	Collaboration	5 898	20 983	1 130	0.03	2 085	18
CAIDA (CAI)	Autonomous Sys.	6 855	13 341	1 650	0.02	1 412	18
COM-YOUTUBE (CYT)	Social	452 060	2 295 072	66 200	5 090	4 136	18
DENMARK (DNK)	Road	252 416	320 914	147 000	0.75	13 152	18
FLICKREDGES (FED)	Social	105 512	2 316 450	2 180	165	12 415	18
FORESTFIRE (FF2)	Synthetic	1 178 888	13 849 776	12 500	16 100	17 983	18
FLICKRLINKS (FLI)	Social	704 985	14 501 930	125 000	17 700	12 525	18
OREGON (ORE)	Autonomous Sys.	7 218	19 448	638	2.54	286	18
SKITTER (SKI)	Autonomous Sys.	1 443 769	10 830 987	197 000	11 100	10 666	18
WIKIVOTE (WIK)	Hyperlinks	4 786	98 456	751	1.12	1 890	18
WIKITALK (WIT)	Collaboration	622 315	2 889 703	47 700	38 700	2 951	18

Table 1: Input details and performance of PLL and KHL w.r.t. time and space, resp.

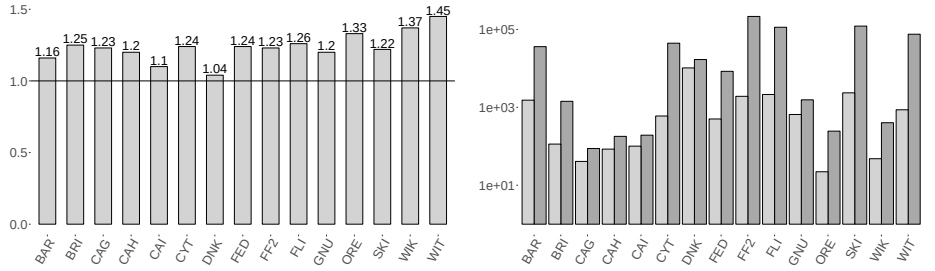
(denoted by KHL in the following) and measured the computational time spent, in seconds, to build $P(G)$ and a 1-EFTPL $P(G,1)$, resp., which is shown in Table 1 (4th and 5th column, resp.). For the sake of completeness, we also measured the average size of $P(G)$ and the overhead required for storing $T(G,1)$, in bytes, per vertex. Such values are reported in Table 1 as well (6th and 7th column, resp.). Finally, we selected 10 000 pairs of vertices of the graph and, for each pair s,t, we executed the following steps:[8] i) we randomly removed an edge e of the graph which, with probability $p = 5/100$ is chosen to be on the shortest path between s and t in G;[9] ii) we ran, on $G-e$, a modified version of Algorithm 1 that includes a *point-of-failure (POI) rerouting procedure*, with s and t as input, and measured both the time taken and the length of the output backup path; notice that, a POI rerouting procedure is a common backup method to deal with failures, that essentially retrieve a feasible solution starting from the point where a failure was encountered [19]; in our case, when one of the two sub-paths computed by Algorithm 1 happens to encounter a failed edge, say for instance the path from s fails at edge $e = (x,y)$, then we build a BFS in $G-e$ rooted at x and we stop as soon as t is reached; iii) we ran Algorithm 3 on $G-e$ with s and t as input, and measured both the time taken and the length of the output backup path; iv) we compute the shortest-path between s and t on $G-e$ (e.g. by means of a BFS) and measure the length of the true shortest path between s and t in $G-e$. We accordingly compute, for pair s,t, the stretch of the path computed by ii) and that of the path computed by iii). For all the considered performance

[7] see http://snap.stanford.edu

[8] All our software has been compiled with GNU g++ v.5 (O4 opt. level) under Linux and executed on an Intel Xeon© CPU

[9] the choice of p is inspired by previous works on failures in networks [14]

metrics, we computed average values over the 10 000 executions. The results concerning the query time (in seconds) are reported in Fig. 1b while those related to the average stretch are shown in Fig. 1a. We have omitted the stretch of the point-of-failure strategy, since it was always negligible, close to 1. Regarding the root vertex r of the $P(G, 1)$, note that, from a worst-case point of view, the stretch provided by the approach is independent from the specific chosen vertex, hence it could be selected at random. However, it is clear that the best choice of the root from a practical viewpoint should be based on some centrality measure, since this would tend to induce short backup paths [13]. Unfortunately, computing fine-grained centrality measures, such as, e.g. betweenness centrality, can be rather computationally expensive and therefore can (heavily) impact on the preprocessing time of our approach. Thus, when needed, we choose r to be the vertex of highest degree, since vertex degree has been shown to be a quite robust centrality measure that can be computed in linear time [10]. For the sake of completeness, we also considered an approximation of the betweenness centrality based on sampling, which however always produced worse results in terms of stretch, and thus we omitted them.



(a) (b)

Figure 1: (a) Estimation of $\alpha(P(G, 1))$ based on 10 000 measures; (b) Query time of Algorithm 3 (lighter gray) versus query time of Algorithm 1 with POI rerouting.

Analysis. The main outcome of our experimental evaluation concerns both average stretch, provided by the new structure $P(G, 1)$, and query time. Regarding the former, despite the $O(kn)$ worst case upper bound of Lemma 2, the $P(G, 1)$ data structure shows a very good behavior in practice, as the measured average stretch is always pretty close to one, and only slightly larger than the stretch exhibited by the rerouting method, which can be considered a benchmark strategy w.r.t. quality of reported shortest paths. Regarding the latter, Algorithm 3 results in being extremely (orders of magnitude) faster (see Fig. 1b) w.r.t. the rerouting-based strategy, which require to perform BFSs every time a failed edge is met. This suggests the method to be a viable option in practice whenever transient failures need to managed. This comes at the price of: i) a negligible time overhead for enriching the $P(G)$ data structure (i.e. for performing KHL), which can be hundreds of times smaller than the time for executing PLL; ii) a constant amount of extra space (few bytes per vertex) for storing $T(G, 1)$, which can be considered negligible w.r.t. the space per vertex required for storing $P(G)$.

5 Conclusion and future work

In this paper, we have studied 2-*hop cover distance/path-reporting labeling schemes* in the fault-tolerant approximate setting. In particular, we have presented a simple and efficient way of enriching this successful scheme in order to make it *resilient* to the failure of a subset of k edges in the graph, by exploiting the notion of so-called *edge-independent* spanning trees. Depending on k, we have provided different strategies and analyzed the corresponding performance. In addition, to assess the practical effectiveness of our method, we have provided an experimental evaluation, conducted for the case $k = 1$, whose results show the new method to be really competitive when compared with the performance of the benchmark non-fault-tolerant scheme, equipped with a point-of-failure rerouting policy. There are several research directions which might deserve further investigation in this area. For instance, it could be interesting to study whether it is possible to design a scheme with a better guarantee on the stretch w.r.t. the one proposed in this paper. Another interesting direction could be that of extending the experimental evaluation of our new method to higher values of k.

Bibliography

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, pages 349–360. ACM, 2013.

[2] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *arXiv preprint arXiv:1504.05140*, 2015.

[3] D. Bilò, L. Gualà, S. Leucci, and G. Proietti. Multiple-edge-fault-tolerant approximate shortest-path trees. In *Proc. of 33rd Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 47 of *LIPIcs*, pages 18:1–18:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[4] J. Cheriyan and S. Maheshwari. Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs. *Journal of Algorithms*, 9(4):507–537, 1988.

[5] A. Cionini, G. D'Angelo, M. D'Emidio, D. Frigioni, K. Giannakopoulou, A. Paraskevopoulos, and C. D. Zaroliagis. Engineering graph-based models for dynamic timetable information systems. In *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, volume 42 of *OASICS*, pages 46–61. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[7] S. Curran, O. Lee, and X. Yu. Finding four independent trees. *SIAM J. Comput.*, 35(5):1023–1058, 2006.

[8] A. D'Andrea, M. D'Emidio, D. Frigioni, S. Leucci, and G. Proietti. Path-fault-tolerant approximate shortest-path trees. In *Proc. of 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 9439 of *Lecture Notes in Computer Science*. Springer, 2015.

[9] G. D'Angelo, M. D'Emidio, and D. Frigioni. A loop-free shortest-path routing algorithm for dynamic networks. *Theoretical Computer Science*, 516:1–19, 2014.

[10] G. D'Angelo, M. D'Emidio, and D. Frigioni. Distance queries in large-scale fully dynamic complex networks. In *Proc. of 27th International Workshop on Combinatorial Algorithms (IWOCA)*, volume 9843 of *Lecture Notes in Computer Science*, pages 109–121. Springer, 2016.

[11] G. D'Angelo, M. D'Emidio, D. Frigioni, and V. Maurizio. A speed-up technique for distributed shortest paths computation. In *Proc. of 11th International Conference on Computational Science and Its Applications (ICCSA)*, volume 6783 of *Lecture Notes in Computer Science*, pages 578–593. Springer, 2011.

[12] G. D'Angelo, M. D'Emidio, D. Frigioni, and C. Vitale. Fully dynamic maintenance of arc-flags in road networks. In R. Klasing, editor, *Proc. of 11th International Symposium on Experimental Algorithms (SEA)*, volume 7276 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2012.

[13] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *Proc. of 22th Annual European Symposium on Algorithms (ESA)*, volume 8737 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 2014.

[14] G. Iannaccone, C.-n. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an ip backbone. In *Proc. of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 237–242. ACM, 2002.

[15] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. *Inf. Comput.*, 79(1):43–59, 1988.

[16] K. Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[17] Y. Qin, Q. Z. Sheng, and W. E. Zhang. SIEF: efficiently answering distance queries for failure prone graphs. In *Proc. of 18th International Conference on Extending Database Technology (EDBT)*, pages 145–156. OpenProceedings.org, 2015.

[18] J. Roskind and R. E. Tarjan. A note on finding minimum-cost edge-disjoint spanning trees. *Mathematics of Operations Research*, 10(4):701–708, 1985.

[19] N. Santoro. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2006.

[20] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. de Castro Reis, and B. A. Ribeiro-Neto. Efficient search ranking in social networks. In *Proc. of 16th ACM Conference on Information and Knowledge Management (CIKM)*, pages 563–572. ACM, 2007.

[21] X. Zhang and L. Chen. Distance-aware selective online query processing over large distributed graphs. *Data Science and Engineering*, 2(1):2–21, 2017.