Algorithmica
https://doi.org/10.1007/s00453-018-0452-3

# An Efficient Strongly Connected Components Algorithm in the Fault Tolerant Model

**Surender Baswana[1] · Keerti Choudhary[1] · Liam Roditty[2]**

Received: 11 September 2017 / Accepted: 3 May 2018


**Abstract** In this paper we study the problem of maintaining the strongly connected components of a graph in the presence of failures. In particular, we show that given a directed graph $G = (V, E)$ with $n = |V|$ and $m = |E|$, and an integer value $k \geq 1$, there is an algorithm that computes in $O(2^k n \log^2 n)$ time for any set $F$ of size at most $k$ the strongly connected components of the graph $G \backslash F$. The running time of our algorithm is almost optimal since the time for outputting the SCCs of $G \backslash F$ is at least $\Omega(n)$. The algorithm uses a data structure that is computed in a preprocessing phase in polynomial time and is of size $O(2^k n^2)$. Our result is obtained using a new observation on the relation between strongly connected components (SCCs) and reachability. More specifically, one of the main building blocks in our result is a restricted variant of the problem in which we only compute strongly connected components that intersect a certain path. Restricting our attention to a path allows us to implicitly compute reachability between the path vertices and the rest of the graph in time that depends

A preliminary version of this paper appeared in the proceedings of ICALP 2017.

This research was partially supported by Israel Science Foundation (ISF) and University Grants Commission (UGC) of India. The research of the second author was partially supported by Google India under the Google India Ph.D. Fellowship Award.

✉ Keerti Choudhary
keerti@cse.iitk.ac.in

Surender Baswana
sbaswana@cse.iitk.ac.in

Liam Roditty
liam.roditty@biu.ac.il

1 Department of Computer Science and Engineering, IIT, Kanpur, India

2 Department of Computer Science, Bar Ilan University, Ramat Gan, Israel

Published online: 21 May 2018

logarithmically rather than linearly in the size of the path. This new observation alone, however, is not enough, since we need to find an efficient way to represent the strongly connected components using paths. For this purpose we use a mixture of old and classical techniques such as the heavy path decomposition of Sleator and Tarjan (J Comput Syst Sci 26:362–391, 1983) and the classical Depth-First-Search algorithm. Although, these are by now standard techniques, we are not aware of any usage of them in the context of dynamic maintenance of SCCs. Therefore, we expect that our new insights and mixture of new and old techniques will be of independent interest.

**Keywords** Fault tolerant · Directed graph · Strongly connected components

## 1 Introduction

Computing the strongly connected components (SCCs) of a directed graph $G = (V, E)$, where $n = |V|$ and $m = |E|$, is one of the most fundamental problems in computer science. There are several classical algorithms for computing the SCCs in $O(m + n)$ time that are taught in any standard undergraduate algorithms course [14].

In this paper we study the following natural variant of the problem in dynamic graphs. What is the fastest algorithm to compute the SCCs of $G \setminus F$, where $F$ is any set of edges or vertices. The algorithm can use a polynomial size data structure computed in polynomial time for $G$ during a preprocessing phase.

The main result of this paper is:

**Theorem 1** *There is an algorithm that computes the SCCs of $G \setminus F$, for any set $F$ of $k$ edges or vertices, in $O(2^k n \log^2 n)$ time. The algorithm uses a data structure of size $O(2^k n^2)$ computed in $O(2^k n^2 m)$ time for $G$ during a preprocessing phase.*[1]

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

This dynamic model is usually called the fault tolerant model and its most important parameter is the time that it takes to compute the output in the presence of faults. It is an important theoretical model as it can be viewed as a restriction of the deletion only (decremental) model in which edges (or vertices) are deleted one after another and queries are answered between deletions. The fault tolerant model is especially useful in cases where the worst case update time in the more general decremental model is high.

There is wide literature on the problem of decremental SCCs. Recently, in a major breakthrough, Henzinger et al. [25] presented a randomized algorithm with $O(mn^{0.9+o(1)})$ total update time and broke the barrier of $\Omega(mn)$ for the problem. Even more recently, Chechik et al. [11] obtained an improved total running time of $O(m\sqrt{n} \log n)$.

However, these algorithms and in fact all the previous algorithms have an $\Omega(m)$ worst case update time for a single edge deletion. This is not a coincidence. Recent

[1] For $k > \log n$, the time taken by our algorithm is $\omega(n^2)$, compared to which the standard static algorithm that takes $O(m + n)$ time is better, so we will be only restricting to the case when $k \le \log n$.

developments in conditional lower bounds by Abboud and Williams [1] and by Henzinger et al. [24] showed that unless a major breakthrough happens, the worst case update time of a single operation in any algorithm for decremental SCCs is $\Omega(m)$. Therefore, in order to obtain further theoretical understanding on the problem of decremental SCCs, and in particular on the worst case update time it is only natural to focus on the restricted dynamic model of fault tolerant.

In the recent decade several different researchers used the fault tolerant model to study the worst case update time per operation for dynamic connectivity in undirected graphs. Pătraşcu and Thorup [32] presented connectivity algorithms that support edge deletions in this model. Their result was improved by the recent polylogarithmic worst case update time algorithm of Kapron et al. [27]. Duan and Pettie [17,18] used this model to obtain connectivity algorithms that support vertex deletions.

In directed graphs, very recently, Georgiadis et al. [21] considered the problem of SCCs but only for a single edge or a single vertex failure, that is $|F| = 1$. They showed that it is possible to compute the SCCs of $G \setminus \{e\}$ for any $e \in E$ (or of $G \setminus \{v\}$ for any $v \in V$) in $O(n)$ time using a data structure of size $O(n)$ that was computed for $G$ in a preprocessing phase in $O(m + n)$ time. Our result is the first generalized result for any constant size $F$. This comes with the price of an extra $O(\log^2 n)$ factor in the running time, a slower preprocessing time and a larger data structure. In [21], Georgiadis, Italiano and Parotsidis also considered the problem of answering strong connectivity queries after one failure. They show construction of an $O(n)$ size oracle that can answer in constant time whether any two given vertices of the graph are strongly connected after failure of a single edge or a single vertex.

In a previous work [2] we considered the problem of finding a sparse subgraph that preserves single source reachability. More specifically, given a directed graph $G = (V, E)$ and a vertex $s \in V$, a subgraph $H$ of $G$ is said to be a $k$-Fault Tolerant Reachability Subgraph (*k-FTRS*) for $G$ if for any set $F$ of at most $k$ edges (or vertices), a vertex $v \in V$ is reachable from $s$ in $G \setminus F$ if and only if $v$ is reachable from $s$ in $H \setminus F$. In [2] we proved that there exists a *k-FTRS* for $s$ with at most $2^k n$ edges.

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

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

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

This problem, however, is much easier since the vertices in the query reveal which two *k-FTRS* we need to scan. In the challenge that we address in this paper *all* the SCCs of $G \setminus F$, for an arbitrary set $F$, have to be computed. However, using the same data structure as before, it is not really clear a-priori which of the *k-FTRS* we need to scan. We note that our algorithm uses the *k-FTRS* which seems to be an essential tool but is far from being a sufficient one and more involved ideas are required.

### 1.1 An Overview of Our Result

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

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

To solve this restricted version, we implicitly solve the problem of reachability from $x$ (and to $x$) in $G \backslash F$, for each $x \in P$. Though it is trivial to do so in time $O(2^k n|P|)$ using *k-FTRS* of each vertex on $P$, our goal is to perform this computation in $O(2^k n \log n)$ time, that is, in running time that is *independent* of the length of $P$ (up to a logarithmic factor). For this we use a careful insight into the structure of reachability between $P$ and $V$. Specifically, if $v \in V$ is reachable from $x \in P$, then $v$ is also reachable from any predecessor of $x$ on $P$, and if $v$ is not reachable from $x$, then it cannot be reachable from any successor of $x$ as well. Let $w$ be any vertex on $P$, and let $A$ be the set of vertices reachable from $w$ in $G \backslash F$. Then we can split $P$ at $w$ to obtain two paths: $P_1$ and $P_2$. We already know that all vertices in $P_1$ have a path to $A$, so for $P_1$ we only need to focus on set $V \backslash A$. Also the set of vertices reachable from any vertex on $P_2$ must be a subset of $A$, so for $P_2$ we only need to focus on set $A$. This suggests a divide-and-conquer approach which along with some more insight into the structure of *k-FTRS* helps us to design an efficient algorithm for computing all the SCCs that intersect $P$.

In order to use the above result to compute all the SCCs of $G \backslash F$, we need a clever partitioning of $G$ into a set of vertex disjoint paths. A Depth-First-Search (DFS) tree plays a crucial role here as follows. Let $P$ be any path from root to a leaf node in a DFS tree $T$. If we compute the SCCs intersecting $P$ and remove them, then the remaining SCCs must be contained in subtrees hanging from path $P$. So to compute the remaining SCCs we do not need to work on the entire graph. Instead, we need to work on each subtree. In order to pursue this approach efficiently, we need to select path $P$ in such a manner that the subtrees hanging from $P$ are of small size. The heavy path decomposition of Sleator and Tarjan [36] helps to achieve this objective.[2]

Our algorithm and data structure can be extended to support insertions as well. More specifically, we can report the SCCs of a graph that is updated by insertions and deletions of $k$ edges in the same running time. The size of our data structure and the preprocessing time remains the same when modified to support edge insertions.

### 1.2 Related Work

The problem of maintaining the SCCs of a graph was studied in the decremental model. In this model the goal is to maintain the SCCs of a graph whose edges are being deleted

[2] We note that the heavy path decomposition was also used in the fault tolerant model in STACS'10 paper of [28], but in a completely different way and for a different problem.

by an adversary. The main parameters in this model are the worst case update time per an edge deletion and the total update from the first edge deletion until the last. Frigioni et al. [20] presented an algorithm that has an *expected* total update time of $O(mn)$ if all the deleted edges are chosen at random. Roditty and Zwick [35] presented a Las-Vegas algorithm with an *expected* total update time of $O(mn)$ and *expected* worst case update time for any single edge deletion of $O(m)$. Łacki [29] presented a deterministic algorithm with a total update time of $O(mn)$, and thus solved the open problem posed by Roditty and Zwick in [35]. However, the worst case update time per a single edge deletion of his algorithm is $O(mn)$. Roditty [34] improved the worst case update time of a single edge deletion to $O(m \log n)$. Recently, in a major breakthrough, Henzinger et al. [25] presented a randomized algorithm with $O(mn^{0.9+o(1)})$ total update time. Very recently, Chechik et al. [11] obtained a total update time of $O(m\sqrt{n}\log n)$. Note that all the previous works on decremental SCC are with $\Omega(m)$ worst case update time. Whereas, our result directly implies $O(n\log^2 n)$ worst case update time as long as the total deletion length is constant.

Most of the previous work in the fault tolerant model is on variants of the shortest path problem. Demetrescu et al. [15] designed an $O(n^2 \log n)$ size data structure that can report the distance from $u$ to $v$ avoiding $x$ for any $u, v, x \in V$ in $O(1)$ time. Bernstein and Karger [4] improved the preprocessing time of [15] to $O(mn \operatorname{polylog} n)$. Duan and Pettie [19] extended the result of [15] to dual failures by designing a data structure of $O(n^2 \log^3 n)$ space that can answer any distance query upon two failures in $O(\log n)$ time. Weimann and Yuster [37] considered the question of optimizing the preprocessing time using Fast Matrix Multiplication (FMM) for graphs with integer weights from the range $[-M, M]$. Grandoni et al. [22] improved the result of [37] based on a novel algorithm for computing all the replacement paths from a given source vertex in the same running time as solving APSP in directed graphs.

For all-pairs approximate distances, Baswana and Khanna [3] showed that for any positive integer $t$, an unweighted undirected graph can be processed to compute an oracle that can report $(2t-1)(1+\epsilon)$-approximate distances between any two nodes upon failure of a vertex in $O(t)$ time. The size of their data structure is $O\big(t^5 n^{1+1/t}(1/\epsilon^4)\log^3 n\big)$. For multiple edge failures in weighted graphs, Chechik et al. [13] showed that if $W$ is the ratio of the heaviest and the lightest weight edge in the graph, then we can compute an oracle of $O\big(k \cdot tn^{1+1/t}\log(nW)\big)$ size that after any $k$ failures can report $(8t-2)(k+1)$-stretched distances in $\widetilde{O}(k \log\log W)$ time. Later Chechik et al. [10] improved this result to obtain $(1+\epsilon)$-approximation at the expense of bigger data structure, for any arbitrary $\epsilon$. The size of their data structure is $O\big(kn^2 \log W(\log n/\epsilon)^k\big)$ and the query time is $\widetilde{O}(k^5 \log\log W)$. The questions of finding graph spanners in the fault tolerant model were studied in [5,9,12,13,16].

For the problem of single source shortest paths Parter and Peleg [30] showed that for unweighted graphs we can compute a subgraph with $O(n^{3/2})$ edges that preserves the distances from source after single failure. They also showed a matching lower bound. For dual failures, Parter [31] and Gupta et al. [23] showed that we can compute a corresponding subgraph with $O(n^{5/3})$ edges. It is also known that the bound of $O(n^{5/3})$ is tight [31]. Recently, Bodwin et al. [8] extended this result to $k$ faults by showing graph constructions with $\widetilde{O}(kn^{2-1/2^k})$ edges.

Concerning approximate post-failure distances, Baswana and Khanna [3] showed that for any unweighted undirected graph we can compute an oracle of $O(n \log n + n/\epsilon^3)$ size that after any vertex failure can report $(1+\epsilon)$-approximate distances from source in $O(1)$ time. For any single edge failure in weighted graphs, Bilò et al. [6] showed that we can construct an oracle of $O(n)$ size which can report 2-approximate distances from the source in $O(1)$ time. They also showed that we can compute an oracle of $O\big((n/\epsilon)\log(1/\epsilon)\big)$ size that can report $(1+\epsilon)$-stretched distances from source in $O\big((\log n/\epsilon)\log(1/\epsilon)\big)$ time. For multiple edge failures, Bilò et al. [7] showed that that we can compute an oracle of $O(kn\log^2 n)$ size that after any $k$ edge failures is able to report the $(2k+1)$-stretched distance from source in $O(k^2\log^2 n)$ time.

### 1.3 Organization of the Paper

We describe notations, terminologies, some basic properties of DFS, heavy-path decomposition, and *k-FTRS* in Sect. 2. In Sect. 3, we describe the fault tolerant algorithm for computing the strongly connected components intersecting any path. We present our main algorithm for handling $k$ failures in Sect. 4. The details on how to extend our algorithm and data structure to support insertions as well is provided in Sect. 5.

## 2 Preliminaries

Let $G = (V, E)$ denote the input directed graph on $n = |V|$ vertices and $m = |E|$ edges. We assume that $G$ is strongly connected, since if it is not the case, then we may apply our result to each strongly connected component of $G$. We first introduce some notations that will be used throughout the paper.

- $T$: A DFS tree of $G$.
- $T(v)$: The subtree of $T$ rooted at a vertex $v$.
- $Path(a, b)$: The tree path from $a$ to $b$ in $T$, assuming $a$ is an ancestor of $b$.
- $depth(Path(a, b))$: The depth of vertex $a$ in $T$.
- IN- EDGES$(v, H)$: The set of all incoming edges to $v$ in graph $H$.
- $G^R$: The graph obtained by reversing all the edges in graph $G$.
- $H(A)$: The subgraph of a graph $H$ induced by the vertices of subset $A$.
- $H \setminus F$: The graph obtained by deleting the edges in set $F$ from graph $H$.
- $SCC_H(v)$: The unique SCC in graph $H$ that contains vertex $v$.
- $P::Q$ : The path formed by concatenating paths $P$ and $Q$ in $G$. Here it is assumed that the last vertex of $P$ is the same as the first vertex of $Q$.
- $P[a, b]$: The subpath of path $P$ from vertex $a$ to vertex $b$, assuming $a$ and $b$ are in $P$ and $a$ precedes $b$.

Our algorithm for computing SCCs in a fault tolerant environment crucially uses the concept of a $k$-fault tolerant reachability subgraph ($k$-FTRS) which is a sparse subgraph that preserves reachability from a given source vertex even after the failure of at most $k$ edges in $G$. A $k$-FTRS is formally defined as follows.

**Definition 1** (*k-FTRS*) Let $s \in V$ be any designated source. A subgraph $H$ of $G$ is said to be a $k$-Fault Tolerant Reachability Subgraph (*k-FTRS*) of $G$ with respect to $s$ if for any subset $F \subseteq E$ of $k$ edges, a vertex $v \in V$ is reachable from $s$ in $G \backslash F$ if and only if $v$ is reachable from $s$ in $H \backslash F$.

In [2], we present the following result for the construction of a *k-FTRS* for any $k \geq 1$.

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

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

- $\mathcal{G}(v)$: The *k-FTRS* of graph $G$ with $v$ as source obtained by Theorem 2.
- $\mathcal{G}^R(v)$: The *k-FTRS* of graph $G^R$ with $v$ as source obtained by Theorem 2.

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

**Lemma 1** *Let $s$ be any designated source and $H$ be a $k$-FTRS of $G$ with respect to $s$. Let $A$ be a subset of $V$ containing $s$ such that every path from $s$ to any vertex in $A$ is contained in $G(A)$. Then $H(A)$ is a $k$-FTRS of $G(A)$ with respect to $s$.*

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

The next lemma is an adaptation of Lemma 10 from Tarjan's classical paper on Depth First Search [33] to our needs.

**Lemma 2** *Let $T$ be a DFS tree of $G$. Let $a, b \in V$ be two vertices without any ancestor-descendant relationship in $T$, and assume that $a$ is visited before $b$ in the DFS traversal of $G$ corresponding to tree $T$. Every path from $a$ to $b$ in $G$ must pass through a common ancestor of $a$ and $b$ in $T$.*

*Proof* Let us assume on the contrary that there exists a path $P$ from $a$ to $b$ in $G$ that does not pass through any common ancestor of $a, b$ in $T$. Let $z$ be the LCA of $a, b$ in $T$, and $w$ be the child of $z$ lying on $Path(z, a)$ in $T$. See Fig. 1. Let $A$ be the set of vertices which are either visited before $w$ in $T$ or lie in the subtree $T(w)$, and $B$ be the set of remaining vertices. Thus $a$ belongs to set $A$, and $b$ belongs to set $B$. Let $x$

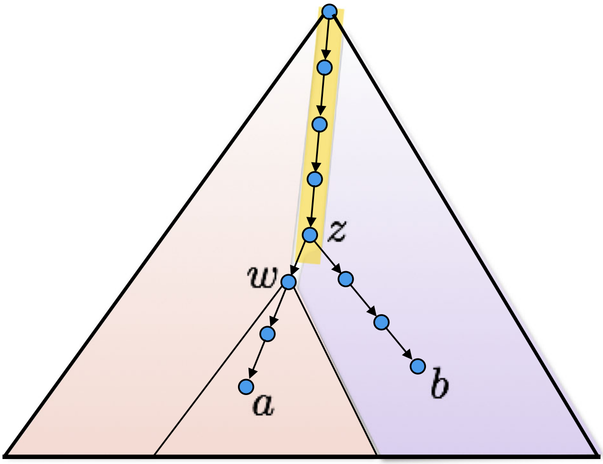


**Fig. 1** Depiction of vertices $a$, $b$, $z$, $w$ and sets $A$ (shown in orange) and $B$ (shown in purple) (Color figure online)

be the last vertex in $P$ that lies in set $A$, and $y$ be the successor of $x$ on path $P$. Thus the edge $(x, y)$ must belong to set $A \times B$. Now all the out-neighbors of $x$ including $y$ must be visited before the DFS traversal finishes for vertex $x$. This along with the fact that $x$ cannot be an ancestor of $w$ implies that vertex $y$ must be visited before the DFS traversal finishes for vertex $w$. This is a contradiction since $y$ lies in $B$, the DFS traversal for vertices in $B$ starts after the DFS traversal finishes for vertex $w$.

### 2.1 A Heavy Path Decomposition

The heavy path decomposition of a tree was designed by Sleator and Tarjan [36] in the context of dynamic trees. This decomposition has been used in a variety of applications since then. Given any rooted tree $T$, this decomposition splits $T$ into a set $\mathcal{P}$ of vertex disjoint paths with the property that any path from the root to a leaf node in $T$ can be expressed as a concatenation of at most $1 + \log n$ subpaths of paths in $\mathcal{P}$, joined together with at most $\log n$ edges. This decomposition is carried out as follows. Starting from the root, we follow the path downward such that once we are at a node, say $v$, the next node traversed is the child of $v$ in $T$ whose subtree is of maximum size, where the size of a subtree is the number of nodes it contains. We terminate upon reaching a leaf node. Let $P$ be the path obtained in this manner. If we remove $P$ from $T$, we are left with a collection of subtrees each of size at most $n/2$. Each of these trees hangs from $P$ through an edge in $T$. We carry out the decomposition of these trees recursively. The following lemma is immediate from the construction of a heavy path decomposition.

**Lemma 3** *For any vertex $v \in V$, the number of paths in $\mathcal{P}$ which start from either $v$ or an ancestor of $v$ in $T$ is at most $1 + \log n$.*

We now introduce the notion of ancestor path.

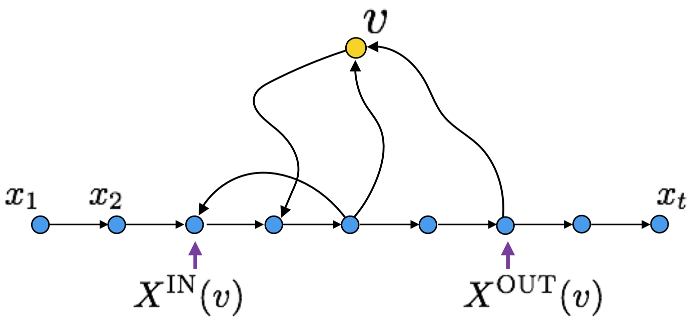


**Fig. 2** Depiction of $X^{\mathrm{IN}}(v)$ and $X^{\mathrm{OUT}}(v)$ for a vertex $v$ whose SCC intersects $X$

**Definition 2** A path $Path(a_1, b_1) \in \mathcal{P}$ is said to be an ancestor path of $Path(a_2, b_2) \in \mathcal{P}$, if $a_1$ is an ancestor of $a_2$ in $T$.

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

## 3 Computation of SCCs Intersecting a Given Path

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

For each $v \in V$, let $X^{\mathrm{IN}}(v)$ be the vertex of $X$ of minimum index (if exists) that is reachable from $v$ in $G\backslash F$. Similarly, let $X^{\mathrm{OUT}}(v)$ be the vertex of $X$ of maximum index (if exists) that has a path to $v$ in $G\backslash F$. (See Fig. 2).

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

**Lemma 4** *For any vertex $w \in V$, the SCC that contains $w$ in $G\backslash F$ intersects $X$ if and only if the following two conditions are satisfied.*

(i) *Both $X^{\mathrm{IN}}(w)$ and $X^{\mathrm{OUT}}(w)$ are defined, and*
(ii) *Either $X^{\mathrm{IN}}(w) = X^{\mathrm{OUT}}(w)$, or $X^{\mathrm{IN}}(w)$ appears before $X^{\mathrm{OUT}}(w)$ on $X$.*

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

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

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

**Lemma 5** *Let $a, b$ be any two vertices in $V$ whose SCCs intersect $X$. Then $a$ and $b$ lie in the same SCC if and only if $X^{\text{IN}}(a) = X^{\text{IN}}(b)$ and $X^{\text{OUT}}(a) = X^{\text{OUT}}(b)$.*

*Proof* In the proof of Lemma 4, we show that if SCC of $w$ intersects $X$, then $X^{\text{IN}}(w)$ and $X^{\text{OUT}}(w)$ are precisely the first and last vertices on $X$ that lie in the SCC of $w$. Since SCCs forms a partition of $V$, vertices $a$ and $b$ will lie in the same SCC if and only if $X^{\text{IN}}(a) = X^{\text{IN}}(b)$ and $X^{\text{OUT}}(a) = X^{\text{OUT}}(b)$.

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

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

**Algorithm 1:** Binary-Search$(i, j, A)$

**if** $(i = j)$ **then**
    **foreach** $v \in A$ **do** $X^{\text{OUT}}(v) = x_i$
**else**
    $mid \leftarrow \lceil (i+j)/2 \rceil$;
    $B \leftarrow$ Reach$(x_{mid}, A)$; /* vertices in $A$ reachable from $x_{mid}$ */
    Binary-Search$(i, mid\text{-}1, A \backslash B)$;
    Binary-Search$(mid, j, B)$;
**end**

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

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

We now explain the execution of function Binary-Search($i, j, A$). If $i = j$, then we assign $x_i$ to $X^{\text{OUT}}(v)$ for each $v \in A$ as justified by Assertion 1. Let us consider the case when $i \neq j$. In this case we first compute the index $mid = \lceil (i + j)/2 \rceil$. Next we compute the set $B$ consisting of all the vertices in $A$ that are reachable from $x_{mid}$. This set is computed using the function Reach($x_{mid}, A$) which is explained later in Sect. 3.2. As follows from Assertion 1, $X^{\text{OUT}}(v)$ for each vertex $v \in A$ must belong to path $(x_i, \ldots, x_j)$. Thus, $X^{\text{OUT}}(v)$ for all $v \in B$ must lie on path $(x_{mid}, \ldots, x_j)$, and $X^{\text{OUT}}(v)$ for all $v \in A \backslash B$ must lie on path $(x_i, \ldots, x_{mid\text{-}1})$. So for computing $X^{\text{OUT}}(\cdot)$ for vertices in $A \setminus B$ and $B$, we invoke the functions Binary-Search($i, mid$-1$, A \backslash B$) and Binary-Search($mid, j, B$), respectively.

### 3.1 Proof of Correctness of Algorithm

In this section we prove that Assertion 1 holds for each call of the Binary-Search function. We also show how this assertion implies that $X^{\text{OUT}}(v)$ is correctly computed for every $v \in V$.

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

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

### 3.2 Implementation of Function Reach

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

**Lemma 6** *If* Binary-Search$(i, j, A)$ is called and $\ell \in [i, j]$, then for each path $P$ from $x_\ell$ to a vertex $z \in A$ in graph in $G \backslash F$, all the vertices of $P$ must be in the set $A$.

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

Lemma 6 and Lemma 1 imply that in order to find the vertices in $A$ that are reachable from $x_{mid}$, it suffices to do a traversal from $x_{mid}$ in the graph $G_A$, the induced subgraph of $A$ in $\mathcal{G}(x) \backslash F$, that has $O(2^k|A|)$ edges. Therefore, based on the above discussion, Algorithm 2 given below, is an implementation of function Reach that takes $O(2^k|A|)$ time.

**Algorithm 2:** Reach$(x_{mid}, A)$

$H \leftarrow \mathcal{G}(x_{mid}) \backslash F$;
$G_A \leftarrow (A, \emptyset)$; /* an empty graph */
**foreach** $v \in A$ **do**
  **foreach** $(y, v) \in$ IN-EDGES$(v, H)$ **do**
    **if** $y \in A$ **then** $E(G_A) = E(G_A) \cup (y, v)$
  **end**
**end**
$B \leftarrow$ Vertices reachable from $x_{mid}$ obtained by a BFS or DFS traversal of graph $G_A$;
Return $B$;

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

**Lemma 7** *The total running time of* Binary-Search$(1, t-1, V_1 \backslash V_t)$ is $O(2^k n \log n)$.

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

We conclude with the following theorem.

**Theorem 3** *Let $F$ be any set of at most k failed edges, and $X = \{x_1, \ldots, x_t\}$ be any path in $G\backslash F$. If we have prestored the graphs $\mathcal{G}(x)$ and $\mathcal{G}^R(x)$ for each $x \in X$, then we can compute all the SCCs of $G\backslash F$ which intersect with $X$ in $O(2^k n \log n)$ time.*

## 4 Main Algorithm

In the previous section we showed that given any path $P$, we can compute all the SCCs intersecting $P$ efficiently, if $P$ is intact in $G\backslash F$. In the case that $P$ contains $\ell$ failed edges from $F$ then $P$ is decomposed into $\ell+1$ paths, and we can apply Theorem 3 to each of these paths separately to get the following theorem:

**Theorem 4** *Let $P$ be any path in $G$ such that for each $x \in P$ we have prestored the graphs $\mathcal{G}(x)$ and $\mathcal{G}^R(x)$. Then for any arbitrary set $F$ of at most $k$ edges we can compute the SCCs of $G\backslash F$ that intersect the path $P$ in $O((\ell+1)2^k n \log n)$ time, where $\ell$ ($\ell \leq k$) is the number of edges in $F$ that lie on $P$.*

Now in order to use Theorem 4 to design a fault tolerant algorithm for SCCs, we need to find a family of paths, say $\mathcal{P}$, such that for any $F$, each SCC of $G\backslash F$ intersects at least one path in $\mathcal{P}$. As described in the Sect. 1.1, a heavy path decomposition of DFS tree $T$ serves as a good choice for $\mathcal{P}$. Choosing $T$ as a DFS tree helps us because of the following reason: let $P$ be any root-to-leaf path, and suppose we have already computed the SCCs in $G\backslash F$ intersecting $P$. Then each of the remaining SCCs must be contained in some subtree hanging from path $P$. The following lemma formally states this fact.

**Lemma 8** *Let $F$ be any set of failed edges, and $Path(a, b)$ be any path in the heavy path decomposition $\mathcal{P}$ of the DFS tree $T$. Let $S$ be any SCC in $G\backslash F$ that intersects $Path(a, b)$ but does not intersect any ancestor path of $Path(a, b)$ in $\mathcal{P}$. Then all the vertices of $S$ must lie in the subtree $T(a)$.*

*Proof* Consider a vertex $u$ on $Path(a, b)$ whose SCC $S_u$ in $G\backslash F$ is not completely contained in the subtree $T(a)$. We show that $S_u$ must contain an ancestor of $a$ in $T$, thereby proving that it intersects an ancestor-path of $Path(a, b)$ in $\mathcal{P}$. Let $v$ be any vertex in $S_u$ that is not in the subtree $T(a)$. Let $P_{u,v}$ and $P_{v,u}$ be paths from $u$ to $v$ and from $v$ to $u$, respectively, in $G\backslash F$. From Lemma 2 it follows that either $P_{u,v}$ or $P_{v,u}$ must pass through a common ancestor of $u$ and $v$ in $T$. Let this ancestor be $z$. Notice that all the vertices of $P_{u,v}$ and $P_{v,u}$ must lie in $S_u$. In particular, $z$ must also lie in $S_u$. Moreover, since $v \notin T(a)$ and $u \in T(a)$, their common ancestor $z$ in $T$ is an ancestor of $a$. Since $z \in S_u$ and it is an ancestor of $a$ in $T$, the lemma follows.

Lemma 8 suggests that if we process the paths from $\mathcal{P}$ in the non-decreasing order of their depths, then in order to compute the SCCs intersecting a path $Path(a, b) \in \mathcal{P}$, it suffices to focus on the subgraph induced by the vertices in $T(a)$ only. This is because the SCCs intersecting $Path(a, b)$ that do not completely lie in $T(a)$ would have already been computed during the processing of some ancestor path of $Path(a, b)$.

We preprocess the graph $G$ as follows. We first compute a heavy path decomposition $\mathcal{P}$ of DFS tree $T$. Next for each path $Path(a, b) \in \mathcal{P}$, we use Theorem 4 to construct the data structure for path $Path(a, b)$ and the subgraph of $G$ induced by vertices in $T(a)$. We use the notation $\mathcal{D}_{a,b}$ to denote this data structure. Our algorithm for reporting SCCs in $G \backslash F$ will use the collection of these data structures associated with the paths in $\mathcal{P}$ as follows.

Let $\mathcal{C}$ denote the collection of SCCs in $G \backslash F$ initialized to $\emptyset$. We process the paths from $\mathcal{P}$ in non-decreasing order of their depths. Let $Path(a, b)$ be any path in $\mathcal{P}$ and let $A$ be the set of vertices belonging to $T(a)$. We use the data structure $\mathcal{D}_{a,b}$ to compute SCCs of $G(A) \backslash F$ intersecting $Path(a, b)$. Let these be $S_1, \ldots, S_t$. Note that some of these SCCs might be a part of some bigger SCC computed earlier. We can detect it by keeping a set $W$ of all vertices for which we have computed their SCCs. So if $S_i \subseteq W$, then we can discard $S_i$, else we add $S_i$ to collection $\mathcal{C}$. Algorithm 3 gives the complete pseudocode of this algorithm.

**Algorithm 3:** Compute SCC($G$, $F$)

$\mathcal{C} \leftarrow \emptyset$; `/* Collection of SCCs */`
$W \leftarrow \emptyset$; `/* A subset of V whose SCC have been computed */`
$\mathcal{P} \leftarrow$ A heavy-path decomposition of $T$, where paths are sorted in the non-decreasing order of their depths;
**foreach** $Path(a, b) \in \mathcal{P}$ **do** `/* visited in non-decreasing order of depth */`
  $A \leftarrow$ Vertices lying in the subtree $T(a)$;
  $(S_1, \ldots, S_t) \leftarrow$ SCCs intersecting $Path(a, b)$ in $G(A) \setminus F$ computed using $\mathcal{D}_{a,b}$;
  **foreach** $i \in [1, t]$ **do**
    **if** ($S_i \nsubseteq W$) **then** Add $S_i$ to collection $\mathcal{C}$ and set $W = W \cup S_i$
  **end**
**end**
Return $\mathcal{C}$;

Note that, in the above explanation, we only used the fact that $T$ is a DFS tree, and $\mathcal{P}$ could have been any path decomposition of $T$. We now show how the fact that $\mathcal{P}$ is a heavy-path decomposition is crucial for the efficiency of our algorithm. Consider any vertex $v \in T$. The number of times $v$ is processed in Algorithm 3 is equal to the number of paths in $\mathcal{P}$ that start from either $v$ or an ancestor of $v$. For this number to be small for each $v$, we choose $\mathcal{P}$ to be a heavy path decomposition of $T$. On applying Theorem 4, this immediately gives that the total time taken by Algorithm 3 is $O(k2^k n \log^2 n)$. In the next subsection, we do a more careful analysis to give a bound of $O(2^k n \log^2 n)$.

### 4.1 Analysis of Time Complexity of Algorithm 3

For any path $Path(a, b) \in \mathcal{P}$ and any set $F$ of failing edges, let $\ell(a, b)$ denote the number of edges of $F$ that lie on $Path(a, b)$. It follows from Theorem 4 that the time spent in processing $Path(a, b)$ by Algorithm 3 is $O\big((\ell(a, b)+1) \times 2^k |T(a)| \times \log n\big)$. Hence the time complexity of Algorithm 3 is of the order of

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

In order to calculate this we define a notation $\alpha(v, Path(a, b))$ as $\ell(a, b) + 1$ if $v \in T(a)$, and 0 otherwise, for each $v \in V$ and $Path(a, b) \in \mathcal{P}$. So the time complexity of Algorithm 3 becomes

$$\begin{aligned}
&2^k \log n \times \left( \sum_{Path(a,b)\in\mathcal{P}} (\ell(a, b) + 1) \times |T(a)| \right) \\
&= 2^k \log n \times \left( \sum_{Path(a,b)\in\mathcal{P}} \sum_{v\in V} \alpha(v, Path(a, b)) \right) \\
&= 2^k \log n \times \left( \sum_{v\in V} \sum_{Path(a,b)\in\mathcal{P}} \alpha(v, Path(a, b)) \right)
\end{aligned}$$

Observe that for any vertex $v$ and $Path(a, b) \in \mathcal{P}$, $\alpha(v, Path(a, b))$ is equal to $\ell(a, b)+1$ if $a$ is either $v$ or an ancestor of $v$, otherwise it is zero. Consider any vertex $v \in V$. We now show that $\sum_{Path(a,b)\in\mathcal{P}} \alpha(v, Path(a, b))$ is at most $k+\log n$. Let $P_v$ denote the set of those paths in $\mathcal{P}$ which starts from either $v$ or an ancestor of $v$. Then $\sum_{Path(a,b)\in\mathcal{P}} \alpha(v, Path(a, b)) = \sum_{Path(a,b)\in P_v} \ell(a, b) + 1$. Since the paths in $P_v$ are disjoint, $\sum_{Path(a,b)\in P_v} \ell(a, b)$ is at most $k$, and Lemma 3 implies that the number of paths in $P_v$ is at most $O(\log n)$. This shows that $\sum_{Path(a,b)\in\mathcal{P}} \alpha(v, Path(a, b)) = O(k + \log n) = O(\log n)$, since $k \leq \log n$.

Hence the time complexity of Algorithm 3 becomes $O(2^k n \log^2 n)$. Finally note that for each vertex $v$, if $v$ belongs to $Path(a, b)$ in the heavy path decomposition, then corresponding to $v$ we store the two $k$-FTRS, defined with respect to $v$ as source and sink, on the subgraph of $G$ induced by vertices in $T(a)$. Thus from Theorem 2, the size of our data structure is at most $O(n \times 2^k n)$ time, and the preprocessing time is $O(n \times 2^k mn)$. We thus conclude with the following theorem.

**Theorem 5** *There is an algorithm that computes the SCCs of $G\backslash F$, for any set $F$ of $k$ edges or vertices, in $O(2^k n \log^2 n)$ time. The algorithm uses a data structure of size $O(2^k n^2)$ computed in $O(2^k n^2 m)$ time for $G$ during a preprocessing phase.*

## 5 Extension to Handle Insertion as Well as Deletion of Edges

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

Let $\mathcal{D}$ denote the $O(2^k n^2)$ size data structure, described in Sect. 4, for handling $k$ failures. In addition to $\mathcal{D}$, we store the two *k-FTRS*: $\mathcal{G}(v)$ and $\mathcal{G}^R(v)$ for each vertex $v$ in $G$. Thus the space used remains the same, i.e. $O(2^k n^2)$. Now let $U = (X, Y)$ be the ordered pair of $k$ updates, with $X$ being the set of failing edges and $Y$ being the set of newly inserted edges. We assume $X \cap Y = \emptyset$ and $|X|, |Y| \leq k$. $X \cap Y = \emptyset$ and $|X|, |Y| \leq k$.

**Algorithm 4:** Find-SCCs($U = (X, Y)$)

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

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

**Lemma 9** *Let $H$ be a graph consisting of edge set $Y$, and the k-FTRS $\mathcal{G}(v)$ and $\mathcal{G}^R(v)$, for each $v \in S$. Then $SCC_{H\setminus X}(v) = SCC_{G+U}(v)$, for each $v \in S$.*

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

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

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

So we compute the auxiliary graph $H$ as described in Lemma 9. Note that $H$ contains only $O(k2^k n)$ edges. Next we compute the SCCs of graph $H \setminus X$ using any standard algorithm [14] that runs in time which is linear in terms of the number of edges and vertices. This algorithm will take $O(2^k n \log n)$ time, since $k$ is at most $\log n$. Finally, for each $v \in S$, we check if the $SCC_{H \setminus X}(v)$ has broken into smaller SCCs in $\mathcal{C}$, if so, then we merge all of them into a single SCC. We can accomplish this entire task in a total $O(nk)$ time only. This completes the description of our algorithm. For the pseudocode see Algorithm 4.

We conclude with the following theorem.

**Theorem 6** *For any $n$-vertex directed graph $G$, there exists an $O(2^k n^2)$ size data structure computable in $O(2^k n^2 m)$ time during a preprocessing phase that, given any set $U$ of at most $k$ edge insertions and at most $k$ edge deletions, can report the SCCs of graph $G + U$ in $O(2^k n \log^2 n)$ time.*

## 6 Future Work

In this paper, we give construction of a data structure of $O(2^k n^2)$ size that after any $k$ edge additions and/or deletions can report the SCCs of the new graph in $O(2^k n \log^2 n)$ time. There are two natural extensions of this problem. The first is to extend the result to answer pair-wise queries: Given a set $U$ of $k > 1$ updates, and a pair of vertices $x, y$, report if the vertices $x$ and $y$ are strongly connected in the graph $G + U$. The second extension is to compute a sparse subgraph that preserves strong-connectivity relation among the vertices of a given graph upon failure of any $k$ edges or vertices. It is also interesting to see if one can improve either the preprocessing time or the size of the data-structure presented in this paper for reporting SCCs after $k$ failures.

## References


1. Abboud, A., Williams, V.V.: Popular conjectures imply strong lower bounds for dynamic problems. In: 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, 18–21 October 2014, pp. 434–443 (2014)
2. Baswana, S., Choudhary, K., Roditty, L.: Fault tolerant subgraph for single source reachability: generic and optimal. In: Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, 18–21 June 2016
3. Baswana, S., Khanna, N.: Approximate shortest paths avoiding a failed vertex: near optimal data structures for undirected unweighted graphs. Algorithmica **66**(1), 18–50 (2013)
4. Bernstein, A., Karger, D.: A nearly optimal oracle for avoiding failed vertices and edges. In: Proceedings of the 41st Annual ACM Symposium on Theory of computing, STOC'09, pp. 101–110. ACM, New York (2009)
5. Bilò, D., Grandoni, F., Gualà, L., Leucci, S., Proietti, G.: Improved purely additive fault-tolerant spanners. In: Proceedings on Algorithms-ESA 2015-23rd Annual European Symposium, Patras, Greece, 14–16 September 2015, pp. 167–178 (2015)
6. Bilò, D., Gualà, L., Leucci, S., Proietti, G.: Compact and fast sensitivity oracles for single-source distances. In: 24th Annual European Symposium on Algorithms, ESA 2016, Aarhus, Denmark, 22–24 August 2016, pp. 13:1–13:14 (2016)
7. Bilò, D., Gualà, L., Leucci, S., Proietti, G.: Multiple-edge-fault-tolerant approximate shortest-path trees. In: 33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, Orléans, 17–20 February 2016, pp. 18:1–18:14 (2016)
8. Bodwin, G., Grandoni, F., Parter, M., Williams, V.V.: Preserving distances in very faulty graphs. In: 44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, Warsaw, Poland, 10–14 July 2017, pp. 73:1–73:14 (2017)

9. Braunschvig, G., Chechik, S., Peleg, D., Sealfon, A.: Fault tolerant additive and $(\mu, \alpha)$-spanners. Theor. Comput. Sci. **580**, 94–100 (2015)
10. Chechik, S., Cohen, S., Fiat, A., Kaplan, H.: (1 + epsilon)-approximate $f$-sensitive distance oracles. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, 16–19 January, pp. 1479–1496 (2017)
11. Chechik, S., Hansen, T.D., Italiano, G.F., Lacki, J., Parotsidis, N.: Decremental single-source reachability and strongly connected components in $\mathrm{O}(\mathrm{m}\sqrt{\mathrm{n}})$ total update time. In: IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, Hyatt Regency, New Jersey, USA, 9–11 October 2016, pp. 315–324 (2016)
12. Chechik, S.: Fault-tolerant compact routing schemes for general graphs. Inf. Comput. **222**, 36–44 (2013)
13. Chechik, S., Langberg, M., Peleg, D., Roditty, L.: f-sensitivity distance oracles and routing schemes. Algorithmica **63**(4), 861–882 (2012)
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
15. Demetrescu, C., Thorup, M., Chowdhury, R.A., Ramachandran, V.: Oracles for distances avoiding a failed node or link. SIAM J. Comput. **37**(5), 1299–1318 (2008)
16. Dinitz, M., Krauthgamer, R.: Fault-tolerant spanners: better and simpler. In: Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, 6–8 June 2011, pp. 169–178 (2011)
17. Duan, R., Pettie, S.: Connectivity oracles for failure prone graphs. In: Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5–8 June 2010, pp. 465–474 (2010)
18. Duan, R., Pettie, S.: Connectivity oracles for graphs subject to vertex failures. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, 16–19 January, pp. 490–509 (2017)
19. Duan, R., Pettie, S.: Dual-failure distance and connectivity oracles. In: SODA'09: proceedings of 19th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 506–515 (2009)
20. Frigioni, D., Miller, T., Nanni, U., Zaroliagis, C.D.: An experimental study of dynamic algorithms for transitive closure. ACM J. Exp. Algoritm. **6**, 9 (2001)
21. Georgiadis, L., Italiano, G.F., Parotsidis, N.: Strong connectivity in directed graphs under failures, with applications. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, 16–19 January, pp. 1880–1899 (2017)
22. Grandoni, F., Williams, V.V.: Improved distance sensitivity oracles via fast single-source replacement paths. In: 53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, 20–23 October 2012, pp. 748–757 (2012)
23. Gupta, M., Khan, S.: Multiple source dual fault tolerant BFS trees. In: 44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, Warsaw, Poland, 10–14 July 2017, pp. 127:1–127:15 (2017)
24. Henzinger, M., Krinninger, S., Nanongkai, D., Saranurak, T.: Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, 14–17 June 2015, pp. 21–30 (2015)
25. Henzinger, M., Krinninger, S., Nanongkai, D.: Improved algorithms for decremental single-source reachability on directed graphs. In: 42nd International Colloquium on Automata, Languages, and Programming, ICALP 2015, Kyoto, Japan, 6–10 July 2015, pp. 725–736 (2015)
26. Italiano, G.F.: Finding paths and deleting edges in directed acyclic graphs. Inf. Process. Lett. **28**(1), 5–11 (1988)
27. Kapron, B.M., King, V., Mountjoy B.: Dynamic graph connectivity in polylogarithmic worst case time. In: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, 6–8 January 2013, pp. 1131–1142 (2013)
28. Khanna, N., Baswana, S.: Approximate shortest paths avoiding a failed vertex: optimal size data structures for unweighted graphs. In: 27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010, Nancy, France, 4–6 March 2010, pp. 513–524 (2010)

29. Lacki. J.: Improved deterministic algorithms for decremental transitive closure and strongly connected components. In: Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, 23–25 January 2011, pp. 1438–1445 (2011)
30. Parter, M., Peleg, D.: Sparse fault-tolerant BFS trees. In: Proceedings on Algorithms-ESA 2013-21st Annual European Symposium, Sophia Antipolis, France, 2–4 September 2013, pp. 779–790 (2013)
31. Parter, M.: Dual failure resilient BFS structure. In: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastían, Spain, 21–23 July 2015, pp. 481–490 (2015)
32. Patrascu, M., Thorup, M.: Planning for fast connectivity updates. In: Proceedings on 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), 20–23 October 2007, Providence, RI, USA, pp. 263–271 (2007)
33. Robert Endre Tarjan: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
34. Roditty, L.: Decremental maintenance of strongly connected components. In: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, 6–8 January 2013, pp. 1143–1150 (2013)
35. Roditty, L., Zwick, U.: Improved dynamic reachability algorithms for directed graphs. SIAM J. Comput. **37**(5), 1455–1471 (2008)
36. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**, 362–391 (1983)
37. Weimann, O., Yuster, R.: Replacement paths and distance sensitivity oracles via fast matrix multiplication. ACM Trans. Algorithms **9**(2), 14 (2013)