Fault Tolerant and Fully Dynamic DFS in Undirected Graphs: Simple Yet Efficient

Surender Baswana
Department of Computer Science & Engineering, IIT Kanpur, Kanpur, India
sbaswana@cse.iitk.ac.in

Shiv Gupta
Department of Computer Science & Engineering, IIT Kanpur, Kanpur, India
shivguptamails@gmail.com

Ayush Tulsyan
Department of Computer Science & Engineering, IIT Kanpur, Kanpur, India
ayushtulsyan01@gmail.com

Abstract

We present an algorithm for a fault tolerant Depth First Search (DFS) Tree in an undirected graph. This algorithm is drastically simpler than the current state-of-the-art algorithms for this problem, uses optimal space and optimal preprocessing time, and still achieves better time complexity. This algorithm also leads to a better time complexity for maintaining a DFS tree in a fully dynamic environment.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Depth first search, DFS, Dynamic graph algorithms, Fault tolerant

Digital Object Identifier 10.4230/LIPIcs.MFCS.2019.65

Related Version A full version of the paper is available at `https://arxiv.org/abs/1810.01726`.

Funding *Surender Baswana*: This research work was partly done during a visit to University of Paderborn supported by a fellowship from the Alexander von Humboldt Foundation.

1 Introduction

Depth First Search (DFS) is a widely popular graph traversal method. The traversal routine, formalized by Tarjan [44] in 1972, has played a crucial role in various graph problems including reachability, bi-connectivity, topological sorting, and strongly connected components.

Given an undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, DFS traversal on the graph takes $\mathcal{O}(m + n)$ time and results in a DFS tree of G.

Most of the graphs in real-world applications keep changing with time. Vertices and edges keep entering and leaving the graph at various time steps. This dynamic aspect has motivated researchers to design algorithms that can update the solution of the corresponding problem efficiently after each such change in the graph. There are two models used for solving these graph problems, namely, fault tolerant algorithms and dynamic graph algorithms.

The *fault tolerant* version of any problem $\mathcal{P}$ on a graph G is to construct a compact data structure, using which, for any given set of failed edges or vertices F, one can efficiently report the solution of $\mathcal{P}$ on $G \setminus F$. Many elegant fault tolerant algorithms have been designed for problems including connectivity [13, 21, 25], shortest paths [10, 11, 16, 20, 28], and spanners [12, 15].

The *dynamic* version of any problem $\mathcal{P}$ on G is modeled as follows. For any online sequence of updates (insertion or deletion of an edge/vertex), one has to report the solution of $\mathcal{P}$ efficiently after every update. Note that, unlike the fault tolerant version, the updates are persistent in dynamic version, i.e., after each update, the solution has to be reported

taking into account all the updates made so far. Algorithms which handle both insertion and deletion of vertices/edges are called *fully dynamic* graph algorithms, whereas the algorithms that handle either insertions or deletions are called *partially dynamic* graph algorithms, more specifically *incremental* or *decremental* graph algorithms, respectively. The prominent results for dynamic graph problems include connectivity [22, 30, 32], reachability [39, 41], shortest path [19, 29, 40], matching [6, 9, 43], spanner [8, 26, 38], min cut [45], Even-Shiloach tree [23], minimum spanning tree [31, 35], and graph sparsifiers [1].

1.1 Previous Results on Fault Tolerant and Dynamic DFS

Franciosa *et al.* [24] presented an incremental algorithm for maintaining a DFS tree in a directed acyclic graph (DAG) which takes overall $\mathcal{O}(mn)$ time for any sequence of m edge insertions. For undirected graphs, Baswana and Khan [7] presented an incremental algorithm for a DFS tree which takes overall $\mathcal{O}(n^2)$ time for any sequence of edge insertions. Baswana and Choudhary [4] designed a randomized decremental algorithm for a DFS tree in a DAG with overall expected $\mathcal{O}(mn \log n)$ time for any sequence of edge deletions.

None of the partially dynamic algorithms stated above achieves an $o(m)$ bound over the worst-case complexity of a single update. Moreover, there were no fully dynamic or fault tolerant algorithms for DFS in undirected graphs until recently. In 2016, the first fault tolerant algorithm was presented that takes $\mathcal{O}(nk \log^4 n)$ time to report a DFS tree for any set of k failed vertices or edges [3]. The time complexity was further improved by Chen *et al.* [17] to $\mathcal{O}(nk \log^2 n)$. Both [3, 17] require a data structure occupying $\mathcal{O}(m \log^2 n)$ bits. Nakamura and Sadakane [34] reduced the space occupied by the data structure to $\mathcal{O}(m \log n)$ bits which is indeed optimal [1]. Using the standard technique of periodic rebuilding, the fault tolerant algorithms presented in [3, 17, 34] were also extended to $o(m)$ fully dynamic DFS algorithms (refer to Table 1 for comparison).

Recently, Chen *et al.* [18] designed an $\mathcal{O}(n)$ time incremental algorithm for DFS tree in undirected graphs, which is optimal if it is required to output the DFS tree after each update. For the hardness results for the dynamic *ordered* DFS problem, the reader may refer to the work of Reif [36, 37] and Miltersen *et al.* [33].

1.2 Familiarizing with the Fault Tolerant DFS Problem

For an undirected graph, DFS traversal results in a spanning tree rooted at the vertex from where the DFS begins. The depth first nature of the traversal ensures the following property:

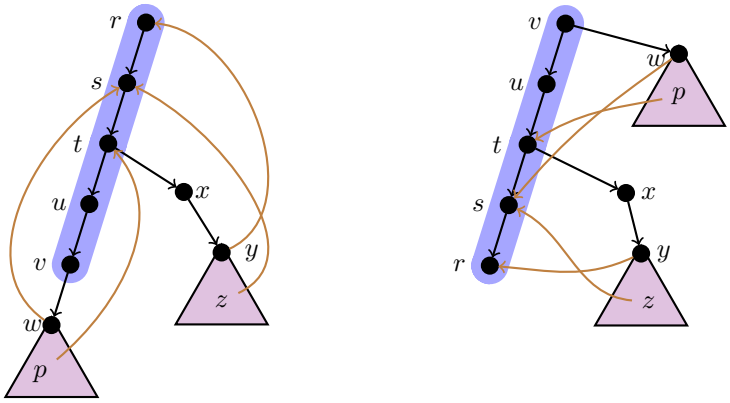
▶ **Property 1.** *(DFS Property) For each vertex $v \in V$, every neighbour of v in the graph appears either as an ancestor of v or as a descendant of v in any DFS tree.*

The DFS property implies that a non-tree edge is never a *cross edge* - an edge whose endpoints do not share an ancestor-descendant relationship. It is due to this reason that each non-tree edge is called a back edge. We now define ancestor-descendant paths.

▶ **Definition 2.** *(Ancestor-descendant Path) A path in a DFS tree is called an ancestor-descendant path if its endpoints have an ancestor-descendant relationship in the tree.*

In order to familiarize with the problem of fault tolerant DFS tree, we discuss another related, but simpler problem, namely, rerooting of a DFS tree defined as follows.

[1] Precisely, their data structure occupies $(m + o(m)) \log n$ bits by using a wavelet tree.



(a) a DFS tree rooted at r.

(b) reversing $path(r, v)$ results in transforming many back edges to cross edges.

Figure 1 Non-triviality of rerooting problem.

▶ **Problem 3.** *Preprocess an undirected graph $G = (V, E)$ to build a compact data structure so that given any vertex $v \in V$, we can report the DFS tree rooted at v efficiently.*

Let T be an initial DFS tree, rooted at a vertex, say r (see Figure 1(a)). We use $T(q)$ to represent the subtree of T rooted at the vertex q. Also, let $path(a, b)$ denote the path from vertex a to b in T. For computing a DFS tree rooted at any vertex v, the first natural idea is to just reverse the direction of the path from r to v in T. However, this may result in transforming many back edges to cross edges and hence a violation of the DFS property (see Figure 1(b)). To fix this problem, we may need to reroot various subtrees hanging from the reversed path. Along these lines, [3] presented an algorithm that takes $\mathcal{O}\left(n \log^3 n\right)$ time to compute a DFS tree rooted at any vertex.

In order to see how rerooting a DFS tree is related to the problem of fault tolerant DFS tree, consider the failure of vertex u in Figure 1(a). The subtree $T(v)$ is connected to the remaining tree through many back edges, and the back edge (p, t) is incident closest to u on the $path(r, u)$. If we reroot the subtree $T(v)$ at vertex p and hang it from the remaining tree through the edge (p, t), the resulting tree will indeed be a valid DFS tree of $G \backslash \{u\}$.

The fault tolerant DFS tree problem becomes more complex in the presence of multiple faults. However, the distribution of faults plays an important role as follows. If the failing vertices do not have any ancestor-descendant relationship in the DFS tree, they can be handled *independently*. For example, the simultaneous failure of vertices u and x in the DFS tree shown in Figure 1(a) requires rerooting the respective subtrees $T(v)$ and $T(y)$ at vertices p and z respectively. So, even for arbitrarily large number of faults, if no two of them appear on the same root to leaf path, we have to just reroot the corresponding disjoint subtrees of T to report the DFS tree avoiding those failures. However, if two or more faults indeed appear on a single root to leaf path, the problem becomes more complex. For this case, [3] presents an algorithm which is quite different from their rerooting algorithm.

1.3 Overview of the Previous Results

We now begin with an overview of the existing algorithms for fault tolerant DFS tree. For any set of k failures, the algorithm presented in [3] first partitions the original DFS tree into a pool of connected components. This pool consists of k paths (specifically, *ancestor-descendant paths*) and potentially a large number of subtrees. The algorithm treats each

of these components as a *super vertex* and uses them to grow the DFS tree T^* that avoids all the failures. At a high level, the algorithm can be visualized as a traversal on these super vertices. Each traversal extracts a path from the super vertex, attaches it to T^*, and places the remaining portion of the super vertex back into the pool. In order to pursue DFS traversal further in an efficient manner, the algorithm needs to compute *minimal* adjacency lists for the vertices of the traversed path (referred to as *reduced adjacency lists*). The algorithm makes use of the following crucial property of DFS traversal.

▶ **Property 4.** *(Components Property [3]) Consider any DFS Traversal on any undirected graph $G = (V, E)$. When the traversal reaches a vertex $v \in V$, let the set of connected components induced by the unvisited vertices be C. If from any component $c \in C$, there exists two edges - e to vertex v and e' to any of the visited vertices (including v), then for building a valid DFS tree, it is sufficient to consider only the edge e during the rest of DFS traversal, that is, e' can be ignored.*

In order to use the above property to populate the reduced adjacency list, the algorithm needs a data structure to answer the following queries repeatedly.

- $Query(w, x, y)$: among all the edges from w that are incident on the $path(x, y)$ in T, return an edge that is incident nearest to x on the $path(x, y)$.
- $Query(T(w), x, y)$: among all the edges from $T(w)$ that are incident on the $path(x, y)$ in T, return an edge that is incident nearest to x on the $path(x, y)$.

It is quite obvious from the description given above that these queries are quite non-trivial, and so a sophisticated data structure is designed in [3] to answer these queries efficiently. In addition to the complex data structure, the complete difference in the processing of a path and a subtree obfuscates the algorithm and its analysis.

The subsequent results [17, 34] keep the algorithm unchanged and replace the data structure used in [3] with alternate data structures. Chen *et al.* [17] model the two queries mentioned above as Orthogonal Range Successor/Predecessor(ORS/ORP) queries and this improves the query processing time. Nakamura and Sadakane [34] compressed the data structure used in [17] using Wavelet Trees [27] to achieve optimal space. Despite these improvements, the core of the fault tolerant algorithm remains intricate and the data structure still remains complex. Recently, in an empirical study [5], it was found that this algorithm, for incremental updates, performs even worse than the static DFS algorithm for certain classes of graphs. This naturally raises the question whethere there exists a simpler algorithm for this fundamental problem.

In addition to being complex, all these algorithms fail to exploit the distribution of the faults to achieve efficiency. The running time of these algorithm is $\mathcal{O}(nk\ polylog(n))$ time irrespective of how the k faults are distributed in the DFS tree. In the extreme case, when no two faults share the ancestor-descendant relaitonship, there is a simple $\mathcal{O}(n\ polylog(n))$ time algorithm as described in Section 1.2. This raises the question whether it is possible to have a fault tolerant DFS algorithm whose time complexity depends upon the maximum number of faults lying on any ancestor-descendant path instead of the total number of faults.

1.4 Our Contribution

We take a much simpler approach as compared to the previous algorithms. We first present a new and simple rerooting algorithm based on the following ideas. After building an initial DFS tree, say T, we decompose T into a disjoint collection $\mathcal{P}$ of ancestor-descendant paths. Similar to [3], each of these paths are treated like *super vertices*. At a high level, the algorithm can still be viewed as a traversal on these super vertices. However, as the reader may also

verify, the algorithm turns out to be lighter and quite different at the core. Interestingly, the original DFS tree alone acts as a powerful data structure to be used for rerooting or for the computation of another valid DFS tree in the presence of faults. The algorithm crucially exploits an implicit hierarchy among the ancestor-descendant paths in $\mathcal{P}$. This hierarchy along with the DFS property of T enables us to use much simpler queries. In particular, each query will ask only for an edge from a vertex to one of its ancestor paths in the hierarchy. The hierarchy allows us to represent T as another tree structure, called *shallow tree*. In a nutshell, our algorithm can be viewed as an efficient DFS traversal guided by this shallow tree.

This rerooting algorithm extends to the fault tolerant algorithm with very little and obvious modifications. While preserving simplicity, the fault tolerant algorithm turns out to be faster than all the previous algorithms. Moreover, our algorithm is the first to implicitly incorporate the distribution of faults to gain efficiency. We summarize our result in the following theorem.

▶ **Theorem 5.** *An undirected graph G can be preprocessed in $\mathcal{O}(m+n)$ time to build a DFS tree, say T, and a data structure of $\mathcal{O}(m+n)$ words[2] such that for any set F of k failed vertices or edges, a DFS tree of $G \setminus F$ can be reported in $\mathcal{O}(n(k'+\log n)\log n)$ time, where $k' \leq k$ is the maximum number of faults on any root-leaf path in the tree T.*

We now present the highlights of our algorithm.
Drastically simpler algorithm: Our algorithm is drastically simpler and more intuitive than the previous algorithm. We feel confident to defend that it can be taught even in an undergraduate course on algorithms. The pseudo-codes in Algorithm 1 and Algorithm 2 are concise and very close to the corresponding implementations.

Faster time complexity: Our algorithm takes $\mathcal{O}(n(k'+\log n)\log n)$ time, where k' is the maximum number of failures on any ancestor-descendant path of the DFS tree when k edges/vertices fail. In the worst-case k' can be as large as k. However, k' can be $o(k)$ as well. In the latter case, our result improves all the existing results significantly. Moreover, even in the case $k' = k$, our time complexity is superior to the previous best by a log factor.

Optimal preprocessing time: Our preprocessing relies upon DFS traversal only, taking $\mathcal{O}(m+n)$ time. Given a graph, in order to report the initial DFS tree, one anyway has to run a static DFS. Hence, our preprocessing time is optimal.

Optimal space and elementary data structure: In contrast to the heavy data structure used by [3, 17], our algorithm makes use of very elementary data structures which are compact as well. Each vertex keeps an array storing edges incident on it from ancestors sorted according to their levels. This data structure uses just $m+n$ words and still achieves $\mathcal{O}\left(n(k'+\log n)\log^2 n\right)$ time to report a DFS tree upon failure of any k vertices or edges. By using fractional cascading [14], we get rid of one log factor while still keeping space requirement to be $\mathcal{O}(m+n)$ words.

Faster Fully Dynamic Algorithm: Using Theorem 5 and periodic rebuilding technique used in [3], we also get the fastest algorithm for fully dynamic DFS.

▶ **Theorem 6.** *Given an undirected graph, one can maintain a DFS tree for any online sequence of insertions and deletions of vertices/edges in $\mathcal{O}\left(\sqrt{mn\log n}\right)$ worst-case time per update.*

[2] One word stores $\lceil \log m \rceil$ bits.

Table 1 Comparison of the existing and the new results. Note that k is the total number of faults, whereas k' is the maximum number of faults lying on any root-to-leaf path of the DFS tree.

	[3]	[17]	[34]	**New**
Space (in bits)	$\mathcal{O}(m \log^2 n)$	$\mathcal{O}(m \log^2 n)$	$\mathcal{O}(m \log n)$	$\mathcal{O}(m \log n)$
Preprocessing	$\mathcal{O}(m \log n)$	$\mathcal{O}(m \log n)$	$\mathcal{O}\left(m\sqrt{\log n}\right)$	$\mathcal{O}(m+n)$
Fault tolerant	$\mathcal{O}(nk \log^4 n)$	$\mathcal{O}(nk \log^2 n)$	$\mathcal{O}(nk \frac{\log^3 n}{\log \log n})$	$\mathcal{O}(n(k' + \log n) \log n)$
Dynamic DFS	$\mathcal{O}(\sqrt{mn} \log^{2.5} n)$	$\mathcal{O}(\sqrt{mn} \log^{1.5} n)$	$\mathcal{O}(\sqrt{mn} \frac{\log^{1.75} n}{\sqrt{\log \log n}})$	$\mathcal{O}(\sqrt{mn \log n})$

The new fully dynamic algorithm can be used to solve the dynamic subgraph problems discussed in [3] and improves upon their time complexity as well. Due to space constraint, we do not discuss these problems here. These can be accessed in the full version of the paper.

Table 1 offers a comparison of our results with all the previous results.

1.5 Organisation of the Paper

Section 2 introduces the notations and some well-known techniques/properties used throughout the paper. Section 3 defines the *shallow tree* representation, a concise structure which encapsulates the hierarchy of paths in the initial DFS tree. Section 4 is the core of our work. Here, we describe how a DFS tree can be rerooted efficiently. Section 5 describes how with some minor modifications to the data structure, rerooting procedure extends to a fault tolerant algorithm. We present the fully dynamic algorithm in Section 6.

2 Preliminaries

2.1 Notations

Following notations will be used throughout this paper.

- T: Any DFS tree of the original graph G.
- $dfn(x)$: The depth first number, i.e., the discovery time of the vertex x during the DFS traversal.
- $v(i)$: the vertex $x \in V$ such that $dfn(x) = i$.
- $dist(x, y)$: distance between the vertices x and y in the DFS tree T.

For the sake of ease of explanation, we shall assume that the graph remains connected at all times. This assumption is without loss of generality because of the following standard way of transforming the original graph right in the beginning - Introduce a dummy vertex r and connect it to all vertices of the graph. Henceforth, we maintain a DFS tree rooted at r for this augmented graph. It is easy to observe that the augmented graph remains connected throughout and the DFS tree rooted at r will be such that the subtrees rooted at the children of r constitute a DFS forest of the original graph.

2.2 Heavy-Light Decomposition

Sleator and Tarjan, in their seminal result on dynamic trees [42] introduced a technique of partitioning any rooted tree called Heavy-light decomposition. Given any rooted tree, this technique splits it into a set of vertex-disjoint ancestor-descendant paths. It marks all the

tree edges either dashed or solid - a tree edge is marked solid iff the subtree of the child vertex is heaviest (in terms of number of vertices) among the subtrees of all its siblings and dashed otherwise. A maximal sequence of vertices connected through solid edges constitutes the required ancestor-descendant path. This decomposition can be carried out using DFS traversal in $\mathcal{O}(n)$ time.

2.3 Fractional Cascading

Given n sorted arrays and a value x, suppose we need to find the predecessor/successor of x in each of them. A naive way is to make a binary search on each array. Chazelle and Guibas [14] introduced a novel tool called fractional cascading using which this problem can be solved more efficiently. Also, Chen *et al.*[18] used this tool for arriving at an $\mathcal{O}(n)$ algorithm for incremental updates. We adapt a customized version of their method.

▶ **Lemma 7.** *Fractional cascading: Given n sorted arrays $\{A_i\}_{i\in[n]}$ each with $l_i = |A_i|$ elements and total $\sum_i^n l_i = m$ elements. There exists a data structure of $\mathcal{O}(m+n)$ words, which can be built in $\mathcal{O}(m+n)$ time, such that for any given x, i, and k satisfying $i, k \in [n]$ and $i + k \leq n$, we can search for x (or its predecessor/successor) in all arrays $A_i, \ldots, A_{i+k}$ using the data structure in $\mathcal{O}(k + \log m)$ time.*

3 Shallow Tree Representation

We now introduce the shallow tree representation for DFS tree T that plays a key role in our algorithm. Using heavy-light decomposition, T is broken down into a set of vertex-disjoint ancestor-descendant paths. Let's denote this set with $\mathcal{P}$. Observe that these paths are connected through dashed edges in T. These dashed edges introduce a hierarchy among paths in $\mathcal{P}$ and the shallow tree defined below captures this hierarchy.

▶ **Definition 8.** *Given a DFS tree T of an undirected graph G, let $\mathcal{P}$ be the set of paths obtained through heavy-light decomposition of T. Let H be the set of edges marked dashed during the decomposition. For tree T, its shallow tree S is a rooted tree formed by collapsing each element of $\mathcal{P}$ into a single node (super vertex). Note that, for each edge $(y, z) \in H$ with $y = parent(z)$, the node in S that contains y is the parent of the node containing z.*

Figure 2 demonstrates how a DFS tree is decomposed to form a set of ancestor-descendant paths $\mathcal{P}$ which is subsequently used to form the shallow tree S.

To avoid ambiguity, we address the vertices in the shallow tree as "nodes" and the vertices in the DFS tree as "vertices". $node(x)$ denotes the shallow tree node corresponding to the path in $\mathcal{P}$ containing vertex x.

The construction of S described above ensures the following simple but crucial properties.

- As a result of heavy-light decomposition of a tree T with n vertices, there can be at most $\log n$ dashed edges on any root to leaf path in T. Recall that each edge in S corresponds to a dashed edge. Thus, the depth of any node in tree S can't be larger than $\log n$. It is because of this small depth that we choose the name *shallow tree* for S.
- From the DFS property, neighbours of any vertex $v \in V$ in the graph are either ancestors or descendants of v in T. Consider any such neighbour u. Let p_1 be the path in $\mathcal{P}$ containing v and p_2 be the path in $\mathcal{P}$ containing u. u and v may also lie in the same path in $\mathcal{P}$. From the construction of S, the nodes corresponding to p_1 and p_2 will share an ancestor-descendant relation in S. So we can state the following lemma.

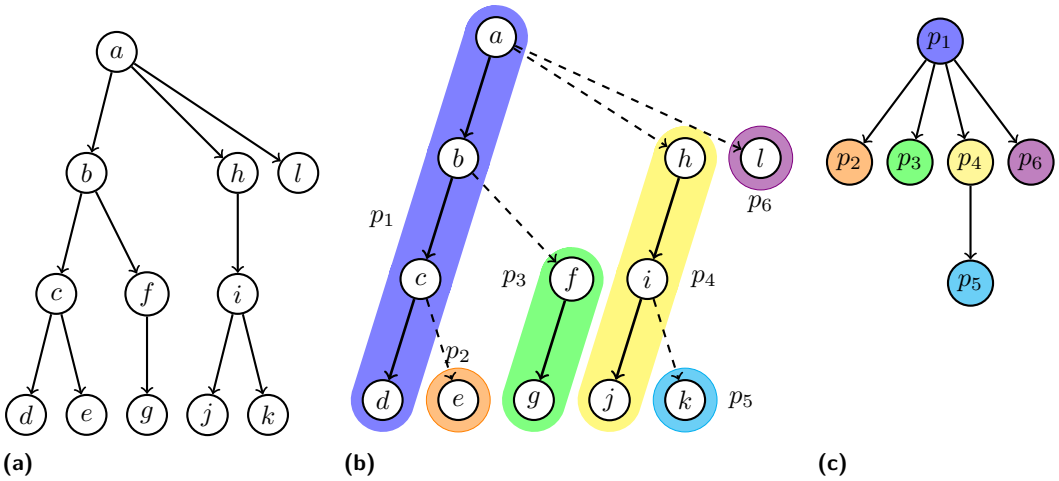


Figure 2 (a) A DFS tree T. (b) Heavy light decomposition of T and the resulting paths in $\mathcal{P}$. (c) The corresponding shallow tree.

▶ **Lemma 9.** *For a DFS tree T of an undirected graph $G = (V, E)$ with shallow tree S, any vertex $v \in V$ which lies in node $\mu \in S$ can have edges only to vertices lying in the nodes which are ancestors or descendants of μ in S.*

We require that the vertices of each solid path have consecutive dfn. This enables us to represent each path $p \in \mathcal{P}$ in a compact manner using just the smallest and largest dfn of vertices on p. For each path p, we store this pair as $PathEndPoints$ at the corresponding node of the shallow tree S. Assigning consecutive dfn to each solid path can be accomplished easily - we carry out another DFS on T where for each vertex, the next vertex to be visited is its child hanging through a solid edge. Thus, the processing of T to make the shallow tree S requires only DFS on T and takes $\mathcal{O}(n)$ time.

4 Rerooting DFS Tree T

Given a DFS tree T for a graph $G = (V, E)$ and a vertex $r' \in V$, the objective is to compute a DFS tree T^* rooted at r' for the same graph G. First we compute the shallow tree representation S of T. We now describe the rerooting procedure.

4.1 Reroot Procedure

The tree T^* is empty in the beginning and is grown gradually starting from r'. To build this tree efficiently, we use the following two ideas. The first idea is to re-use the paths from $\mathcal{P}$. Observe that while rerooting, if we use the original adjacency list for scanning the neighbours, we need $\mathcal{O}(m + n)$ time. So the second idea is to populate for each vertex only a subset of its adjacency list which is small and yet sufficient to compute a DFS tree (this idea was used by [3] in the fault tolerant DFS problem). These lists we refer as *reduced adjacency lists*. We take a lazy and frugal approach to populate them.

The intuition sketched above is materialized by carrying out a DFS traversal *guided* by the shallow tree S. Note that a node of the shallow tree corresponds to a path in $\mathcal{P}$. To compute T^*, our algorithm performs a sequence of steps. Each step begins with entering a node of S through some vertex present on the path stored at the node and leaving it after traversing the path along one direction. The first node to be visited is $node(r')$. We now

provide complete details of the computation involved in each step. Consider any node $\nu \in S$. Let $path(y, z)$ be the path corresponding to ν. When the DFS traversal enters ν through a vertex, say x, the following 3 simple operations are carried out.

1. *Move towards the farther end of the path.*
   We determine the vertex from $\{y, z\}$ farther from x. Let this vertex be y. DFS traversal proceeds from x to y and $path(x, y)$ is attached to the tree T^*. Next, we update the $PathEndpoints\,(\nu)$ such that it stores the endpoints of the untraversed part of $path(y, z)$. This choice of direction ensures that at least half of the path is traversed (referred to as path halving technique, also used in [2, 3]).
2. *Populate the reduced adjacency list of the path just traversed.*
   For vertices on $path(x, y)$, we populate the reduced adjacency list $\mathcal{L}$ using ancestors and descendants of ν in the shallow tree S. This will be explained below in Section 4.1.1.
3. *Continue traversal.*
   Using the reduced adjacency list $\mathcal{L}$ of the traversed path computed in step 2 above, we continue the DFS traversal along the unvisited neighbours of the vertices in the order from y to x (opposite to the direction of traversal, due to the recursive nature of DFS).

Algorithm 1 presents the complete pseudocode of the rerooting procedure based on the above 3 steps. All vertices are marked *unvisited* initially and the reduced adjacency list $\mathcal{L}$ is empty. Invoking Reroot(r') produces the DFS tree rooted at r'.

Algorithm 1 Recursive procedure to reroot the DFS Tree T.

```
1  Function Reroot (x)
2      (y, z) ← PathEndpoints (node (x)) ;
3      if dist(x, z) > dist(x, y) then Swap (y, z);     /* compute dist using dfns */
4      Attach path (x, y) to T*;
5      if x ≠ z then
6          w ← Neighbour of x on path(y, z) nearest to z ;
7          PathEndpoints (node (x)) ← (w, z) ;              /* untraversed path */
8      end
9      L ← ReducedAL (L, (x, y)) ;      /* Update L for vertices on path(x, y) */
10     for i = dfn(y) to dfn(x) do status(v(i)) ← visited ;
11     for i = dfn(y) to dfn(x) do
12         foreach vertex u ∈ L(v(i)) do
13             if status(u) = unvisited then {add (v (i) , u) to T*; Reroot(u)} ;
14         end
15     end
```

4.1.1 Populating Reduced Adjacency Lists

Here we define a query $Q\,(u, (p_s, p_e))$, where $u, p_s, p_e \in V$ satisfy the following constraints:

- p_s and p_e are the endpoints of an ancestor-descendant path in T.
- u is a descendant of the highest vertex on $path\,(p_s, p_e)$, but does not lie on the path.

This query finds an edge from the vertex u that is incident to $path(p_s, p_e)$ closest to p_e. It returns *null* if no such edge exists, otherwise it returns the endpoint other than u.

Consider any node $\nu \in S$, and let $path(y, z)$ be its corresponding path in $\mathcal{P}$. When the DFS traversal enters ν through x and proceeds towards y, we populate the reduced adjacency lists of vertices on $path(x, y)$ using Lemma 9 as follows.

- *Processing Ancestors of vertices on path(x, y).*
  For each $u \in path(x,y)$ and for each ancestor μ of ν, we add $Q(u, PathEndpoints(\mu))$ to $\mathcal{L}(u)$.
- *Processing Descendants of vertices on path(x, y).*
  Using the shallow tree S, one can list all the vertices that are descendants of vertices on $path(x,y)$ in T. From all these vertices, we query for an edge to $path(x,y)$ which is incident closest to y. For any descendant u, if the query $Q(u,(x,y))$ returns a non-*null* value, say w, then we add u to $\mathcal{L}(w)$.

▶ Remark 10. In Algorithm 1, after visiting x, if we moved towards the leaf node, then the untraversed part of ν has to be treated as an ancestor path while populating the reduced adjacency lists of $path(x,y)$, and as a descendant path, otherwise.

Algorithm 2 Populating reduced adjacency lists of vertices on $path(x,y)$.

```
1  Function ReducedAL (L, (x, y))
2      μ ← parent(node(x));
3      while μ ≠ NULL do                          /* edges to ancestor paths */
4          for i = dfn(y) to dfn(x) do
5              L(v(i)) ← L(v(i)) ∪ {Q(v(i), PathEndPoints(μ))};
6          end
7          μ ← parent(μ) ;
8      end
9      if dfn(x) < dfn(y) then z ← x else z ← y;   /* z is ancestor among two */
10     C ← Desc_T(z) \ {v(dfn(x)), ..., v(dfn(y))};
11     foreach u ∈ C do                       /* each descendant of path(x, y) */
12         w ← Q(u, (x, y)) ;                    /* ensure w is closest to y */
13         if w ≠ NULL then L(w) ← L(w) ∪ {u} ;
14     end
15     return L
```

In Algorithm 2, we used query $Q(u,(p_s,p_e))$ as a black box. This query can be answered very easily as follows. The constraints of $Q(u,(p_s,p_e))$ imply that the returned edge is always an edge from u to an ancestor of u. For answering this query, we use a data structure $\mathcal{D}$ which stores the following information for each vertex u.

▶ **Definition 11.** *$\mathcal{D}(u)$ is an array that stores each ancestor of u in T to which u is a neighbour and it stores them in the increasing order of distance from the root of T.*

This data structure enables us to answer query $Q(u,(p_s,p_e))$ using a binary search on array $\mathcal{D}(u)$ and it takes $\mathcal{O}(\log n)$ time only. Interestingly, the data structure $\mathcal{D}$ can be preprocessed very easily in $\mathcal{O}(m+n)$ time as follows. We visit vertices in the increasing order of their depth first numbers. Note that for each vertex v, the neighbours of v which have dfn larger than that of v are its descendants. For each such descendant u, we append v to $\mathcal{D}(u)$. Iterating in increasing order of dfn ensures that all arrays in $\mathcal{D}$ are sorted as needed.

▶ **Theorem 12.** *Given a DFS Tree T of an undirected graph G, it takes $\mathcal{O}(m+n)$ time to build a data structure consisting of exactly $(m+n)$ words which can answer the query $Q(u,(p_s,p_e))$ in $\mathcal{O}(\log n)$ time.*

4.2 Time Complexity Analysis

During preprocessing, we construct the shallow tree S for the DFS tree T and build $\mathcal{D}$. This processing as shown earlier can be completed in $\mathcal{O}(m+n)$ time. The time complexity of the Reroot procedure is bounded by the time required to populate the reduced adjacency lists $\mathcal{L}$. This in turn, is bounded by the number of calls to query Q. To analyse the number of calls made from any vertex w, let ν be the node in the shallow tree S, containing w. In general, if the bound on height of S is d, ReducedAL makes worst-case d queries from w to ancestors of ν. Also when an ancestor of ν in S, say μ, is visited during Reroot procedure, ReducedAL makes a query from w to μ. The path halving technique (line 3 in Algorithm 1) ensures that any node in S (or a path in $\mathcal{P}$) is visited at most $\log n$ times. This implies that any such ancestor μ may be visited at most $\log n$ times. Thus, we can have worst-case $d(\log n+1)$ queries from w to its ancestor paths throughout the Reroot procedure. Summing over all the vertices, there can be at most $nd(\log n+1)$ calls to query Q. Therefore, populating the reduced adjacency lists $\mathcal{L}$ takes overall $\mathcal{O}\left(nd\log^2 n\right)$ time. From Section 3, we know $d \leq \log n$. Thus, using Theorem 12, we can state the following lemma.

▶ **Lemma 13.** *Given a DFS Tree T of an undirected graph G, it takes $\mathcal{O}(m+n)$ time to build a data structure of $(m+n)$ words using which we can compute a DFS Tree of G rooted at any given vertex in $\mathcal{O}\left(n\log^3 n\right)$ time.*

4.2.1 Getting rid of a log factor

Consider the moment when the Reroot procedure enters a $path(y,z)$ through the vertex x and reaches the endpoint y. In Algorithm 2, for each descendant w of $path(x,y)$, we perform query $Q(w,(x,y))$ separately. Instead, using Fractional Cascading, we can perform all these queries together in an efficient manner. Among x and y, let x be the vertex closer to the root of T. As described earlier, all the vertices of $path(x,y)$ have consecutive dfn, and so do the vertices of subtree $T(x)$. Let $last(x)$ be the vertex in $T(x)$ with the largest dfn. Since vertices on $path(x,y)$ have already been visited, we need to query for edges only from vertices with dfn between $dfn(y)+1$ and $dfn(last(x))$. Finding edges to $path(x,y)$ from these vertices can be done with a single query to the fractionally cascaded $\mathcal{D}$ (Lemma 7). It takes $\mathcal{O}(\log m + dfn(last(x)) - dfn(y))$ time to execute this query. We charge the $\log m$ part of the query time to the vertices on $path(x,y)$ and the $dfn(last(x)) - dfn(y)$ is distributed among the descendant vertices. Thus, each descendant vertex incurs a constant charge.

Note that, queries to the ancestors of $path(x,y)$ are answered using the original $\mathcal{D}$ itself. Therefore, in a shallow tree of height d, each vertex $v \in V$ incurs following charges during Reroot procedure - $\mathcal{O}(d\log n)$ when v acts as a descendant in the queries made while visiting ancestors of v, and $\mathcal{O}(d\log n + \log m)$ while visiting v itself. Overall the charge on any vertex is $\mathcal{O}(d\log n)$ and, therefore, the time complexity of Reroot procedure reduces to $\mathcal{O}(nd\log n)$. So we can state the following lemma.

▶ **Lemma 14.** *Given an undirected graph G, the DFS tree T and corresponding shallow tree of height d, it takes $\mathcal{O}(m+n)$ time to build a data structure occupying $\mathcal{O}(m+n)$ words using which the Reroot algorithm executes in $\mathcal{O}(nd\log n)$ time.*

4.3 Correctness of Reroot Procedure

As is evident from the pseudocode of Algorithm 1, besides populating the reduced adjacency lists, Reroot imitates a usual DFS traversal. So in order to show that Reroot computes a valid DFS tree, we need to prove the following. Each edge e that is not added to reduced

adjacency list is indeed redundant and will appear as a back edge in the resulting DFS tree. We can show this as follows.

During Reroot procedure, consider the moment when we attach some path, say $path(x, y)$ to T^*. Lemma 9 implies that all the neighbours of vertices on the $path(x, y)$ will lie either in ancestors or in descendants of $path(x, y)$ in the shallow tree. For each descendant vertex v of $path(x, y)$, we add an edge from v incident on $path(x, y)$ closest to y (lines 12,13 of Algorithm 2). For each ancestor path of $path(x, y)$, say μ, we add an edge from each vertex of $path(x, y)$ to μ (line 5 of Algorithm 2). These ensure that at the moment $path\,(x, y)$ is visited and attached to T^*, from each connected component in the graph induced by unvisited vertices, the edge incident on $path(x, y)$ closest to vertex y is added to $\mathcal{L}$ (if exists). Using Components Property, it follows that all those edges incident on $path(x, y)$, that are not added to $\mathcal{L}$, will appear as back edges in the resulting DFS tree. Hence, the traversal performed using the reduced adjacency lists indeed results in a valid DFS tree.

5 Extension to Fault Tolerant DFS Tree

Given an undirected graph G, we build a DFS tree of G, say T, the corresponding shallow tree S, and the fractionally cascaded D. Let F be the set of failures (edges/vertices) in G, with $|F| = k$. Here, we describe how with some elementary modifications to the shallow tree of T, procedure Reroot can be utilized to report a DFS tree of $G \setminus F$. For the given set F, we update the set $\mathcal{P}$ and the shallow tree S as follows.

1. Each vertex maintains a state: *active* or *failed*. For a failed vertex $x \in F$, we toggle x's state to *failed*. Let $p \in \mathcal{P}$ be the path containing x. We remove x from p. The resulting smaller paths are added to $\mathcal{P}$ and p is removed from $\mathcal{P}$.
2. For each failed edge $e = (u, v)$, we mark the corresponding entries in the adjacency lists of u and v as *failed*. Here, we do not make any changes to fractionally cascaded $\mathcal{D}$. If failed edge e is a tree-edge and was marked *solid* during heavy-light decomposition, $\mathcal{P}$ is updated as follows. Path $p \in \mathcal{P}$ containing e is split into two smaller paths. These smaller paths are added to $\mathcal{P}$ and p is removed from $\mathcal{P}$.
3. After updating $\mathcal{P}$, shallow tree S is updated as follows. For any path $p \in \mathcal{P}$, let the vertex in p closest to root of T be x. The parent of node corresponding to p in S will be the node containing the nearest active ancestor of x.

Given the set F, we can update $\mathcal{P}$ and S in $\mathcal{O}(n)$ time as follows - Make a DFS traversal through T while ensuring vertices are visited in increasing order of their dfn. Update $\mathcal{P}$ and S as discussed above. For each failure, the number of nodes in S increases by at most one. After k updates, S can have at most k new nodes and may not be shallow anymore. However, the depth of a node ν in S doesn't increase due to any failure which does not lie on the path from ν to root of S. Thus, if the maximum number of failures on any root-leaf path in T is k', the height of the shallow tree will be at most $k' + \log n$.

To ensure that no *deleted* vertices/edges are traversed during Reroot procedure, we make the following modifications. Since no changes are made to fractionally cascaded $\mathcal{D}$, the result of some query $Q\,(v, (p_s, p_e))$ may be an endpoint of a *failed* edge. In such a case, we iterate in $\mathcal{D}\,(v)$ starting from the *failed* edge towards p_s until we find an edge present in $G \setminus F$. However, if we cross p_s in doing so, it implies that there is no edge between vertex v and $path\,(p_s, p_e)$, we stop and return *null*. During the procedure Reroot, we spend overall $\mathcal{O}(k)$ time in such iterations. Also note that each node of the updated S corresponds to a path consisting of active vertices only. So $Q\,(v, (p_s, p_e))$ never returns a failed vertex. It follows from the changes in the query and the shallow tree described above that Reroot procedure reports a valid DFS tree of $G \setminus F$. Using Lemma 14, we can conclude the following theorem.

▶ **Theorem 15.** *An undirected graph G can be preprocessed in $\mathcal{O}(m+n)$ time to build a DFS tree, say T, and a data structure of $\mathcal{O}(m+n)$ words such that for any set F of k failed vertices or edges, a DFS tree of $G \setminus F$ can be reported in $\mathcal{O}(n(k'+\log n)\log n)$ time, where $k' \leq k$ is the maximum number of faults on any root-leaf path in the tree T.*

6 Fully Dynamic DFS

We first describe how the fault tolerant DFS algorithm can handle incremental updates. Following that, we use the *overlapped periodic rebuilding technique* to arrive at a fully dynamic DFS algorithm. The ideas utilized in both of these steps were used by Baswana *et al.* [3].

Let U be the set of updates in any undirected graph G. For the edge insertions, we directly add them to the reduced adjacency lists of the endpoints of the edges. To handle insertion of a vertex, we add the new vertices in V and treat their edges as edge insertions. Observe that these modifications are sufficient to handle incremental updates. The size of the reduced adjacency lists after $|U|$ updates is at most $n(|U|+\log n)(\log n+1)$ (from Section 4.2). So the worst-case time complexity of our algorithm is $\mathcal{O}(n(|U|+\log n)\log n)$.

We can use the Reroot procedure to report a DFS tree after every update. But as $|U|$ increases, the algorithm slows down. Therefore, we need to rebuild the data structures (denoted collectively using $\mathcal{A}$) periodically to maintain the efficiency. We rebuild them after, let's say, every c updates. It takes $\mathcal{O}(m+n)$ time to rebuild the data structures. The cost of rebuilding is amortized among the c updates. This results in an amortized runtime fully dynamic algorithm.

To get a worst-case bound, the rebuilding is done in an overlapped fashion as follows. The original data structures $\mathcal{A}_0$ are used till first $2c$ updates. At the end of kc updates (for $k \geq 1$), we start building $\mathcal{A}_k$, the data structures for the graph after incorporating the first kc updates. The computation of building $\mathcal{A}_k$ is distributed evenly over the next c updates to the graph (from $(kc+1)$ to $(k+1)c$ updates). However, during these updates we use $\mathcal{A}_{k-1}$ to report the updated DFS Tree. This ensures $|U|$ is never larger than $2c$. After $(k+1)c$ updates, we have $\mathcal{A}_k$ ready for use and we can discard $\mathcal{A}_{k-1}$. This helps us manage our data structures in $\mathcal{O}(m+n)$ space. For this overlapped periodic rebuilding framework, the following lemma from [3] provides the worst-case bound on the update time we can derive.

▶ **Lemma 16.** *(Overlapped Periodic Rebuilding: Lemma 6.1 in [3]) Let $\mathcal{D}$ be a data structure that can be used to report the solution of a graph problem after a set of U updates on an input graph G. If $\mathcal{D}$ can be built in $\mathcal{O}(f)$ time and the solution for graph $G+U$ can be reported in $\mathcal{O}(h+|U|\cdot g)$time, then $\mathcal{D}$ can be used to report the solution after every update in worst-case $\mathcal{O}\left(\sqrt{fg}+h\right)$ update time, given that $f/g \leq n$.*

Substituting $f=m$, $g=n\log n$ and $h=n\log^2 n$, we obtain, $\mathcal{O}\left(\sqrt{fg}+h\right)=\mathcal{O}\left(\sqrt{mn\log n}\right)$. Hence we conclude with the following theorem.

▶ **Theorem 17.** *An undirected graph can be preprocessed in $\mathcal{O}(m+n)$ time to build a data structure occupying $\mathcal{O}(m+n)$ words, using which one can maintain a DFS tree for any online sequence of insertions and deletions of vertices/edges in $\mathcal{O}\left(\sqrt{mn\log n}\right)$ worst-case time per update.*

References

1 Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On Fully Dynamic Graph Sparsifiers. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 335–344, 2016. doi:10.1109/FOCS.2016.44.

2 Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. Parallel Depth-First Search in General Directed Graphs. *SIAM J. Comput.*, 19(2):397–409, 1990. doi:10.1137/0219025.

3 Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic DFS in Undirected Graphs: breaking the O(m) barrier. In *Symposium on Discrete Algorithms, SODA*, pages 730–739, 2016. doi:10.1137/1.9781611974331.ch52.

4 Surender Baswana and Keerti Choudhary. On Dynamic DFS Tree in Directed Graphs. In *MFCS, Proceedings, Part II*, pages 102–114, 2015. doi:10.1007/978-3-662-48054-0_9.

5 Surender Baswana, Ayush Goel, and Shahbaz Khan. Incremental DFS algorithms: a theoretical and experimental study. In *Symposium on Discrete Algorithms, SODA*, pages 53–72, 2018. doi:10.1137/1.9781611975031.4.

6 Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully Dynamic Maximal Matching in O(log n) Update Time (Corrected Version). *SIAM J. Comput.*, 47(3):617–650, 2018. doi:10.1137/16M1106158.

7 Surender Baswana and Shahbaz Khan. Incremental Algorithm for Maintaining DFS Tree for Undirected Graphs. In *ICALP, Proceedings, Part I*, pages 138–149, 2014. doi:10.1007/978-3-662-43948-7_12.

8 Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4):35:1–35:51, 2012. doi:10.1145/2344422.2344425.

9 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching. *SIAM J. Comput.*, 47(3):859–887, 2018. doi:10.1137/140998925.

10 Davide Bilò, Luciano Gualà, Stefano Leucci, and Guido Proietti. Multiple-Edge-Fault-Tolerant Approximate Shortest-Path Trees. In *33rd Symposium on Theoretical Aspects of Computer Science, STACS*, pages 18:1–18:14, 2016. doi:10.4230/LIPIcs.STACS.2016.18.

11 Davide Bilò, Luciano Gualà, Stefano Leucci, and Guido Proietti. Fault-Tolerant Approximate Shortest-Path Trees. *Algorithmica*, 80(12):3437–3460, 2018. doi:10.1007/s00453-017-0396-z.

12 Gilad Braunschvig, Shiri Chechik, David Peleg, and Adam Sealfon. Fault tolerant additive and (μ, α)-spanners. *Theor. Comput. Sci.*, 580:94–100, 2015. doi:10.1016/j.tcs.2015.02.036.

13 Timothy M. Chan, Mihai Patrascu, and Liam Roditty. Dynamic Connectivity: Connecting to Networks and Geometry. In *Symposium on Foundations of Computer Science, FOCS*, pages 95–104, 2008. doi:10.1109/FOCS.2008.29.

14 Bernard Chazelle and Leonidas J. Guibas. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica*, 1(2):133–162, 1986. doi:10.1007/BF01840440.

15 Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. Fault Tolerant Spanners for General Graphs. *SIAM J. Comput.*, 39(7):3403–3423, 2010. doi:10.1137/090758039.

16 Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. f-Sensitivity Distance Oracles and Routing Schemes. *Algorithmica*, 63(4):861–882, 2012. doi:10.1007/s00453-011-9543-0.

17 Lijie Chen, Ran Duan, Ruosong Wang, and Hanrui Zhang. Improved Algorithms for Maintaining DFS Tree in Undirected Graphs. *CoRR*, abs/1607.04913, 2016. URL: http://arxiv.org/abs/1607.04913v2, arXiv:1607.04913.

18 Lijie Chen, Ran Duan, Ruosong Wang, Hanrui Zhang, and Tianyi Zhang. An Improved Algorithm for Incremental DFS Tree in Undirected Graphs. In *SWAT*, pages 16:1–16:12, 2018. doi:10.4230/LIPIcs.SWAT.2018.16.

19 Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004. doi:10.1145/1039488.1039492.

20 Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. Oracles for Distances Avoiding a Failed Node or Link. *SIAM J. Comput.*, 37(5):1299–1318, 2008. doi:10.1137/S0097539705429847.

21 Ran Duan. New Data Structures for Subgraph Connectivity. In *ICALP, Proceedings, Part I*, pages 201–212, 2010. doi:10.1007/978-3-642-14165-2_18.

22 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. doi:10.1145/265910.265914.

23 Shimon Even and Yossi Shiloach. An On-Line Edge-Deletion Problem. *J. ACM*, 28(1):1–4, 1981. doi:10.1145/322234.322235.

24 Paolo Giulio Franciosa, Giorgio Gambosi, and Umberto Nanni. The Incremental Maintenance of a Depth-First-Search Tree in Directed Acyclic Graphs. *Inf. Process. Lett.*, 61(2):113–120, 1997. doi:10.1016/S0020-0190(96)00202-5.

25 Daniele Frigioni and Giuseppe F. Italiano. Dynamically Switching Vertices in Planar Graphs. *Algorithmica*, 28(1):76–103, 2000. doi:10.1007/s004530010032.

26 Lee-Ad Gottlieb and Liam Roditty. Improved algorithms for fully dynamic geometric spanners and geometric routing. In *Symposium on Discrete Algorithms, SODA*, pages 591–600, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347148.

27 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Symposium on Discrete Algorithms, SODA*, pages 841–850, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

28 Manoj Gupta and Shahbaz Khan. Multiple Source Dual Fault Tolerant BFS Trees. In *44th International Colloquium on Automata, Languages, and Programming, ICALP*, pages 127:1–127:15, 2017. doi:10.4230/LIPIcs.ICALP.2017.127.

29 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time. *J. ACM*, 65(6):36:1–36:40, 2018. URL: https://dl.acm.org/citation.cfm?id=3218657, doi:10.1145/3218657.

30 Monika Rauch Henzinger and Valerie King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *J. ACM*, 46(4):502–516, 1999. doi:10.1145/320211.320215.

31 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

32 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Symposium on Discrete Algorithms, SODA*, pages 1131–1142, 2013. doi:10.1137/1.9781611973105.81.

33 Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity Models for Incremental Computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994. doi:10.1016/0304-3975(94)90159-7.

34 Kengo Nakamura and Kunihiko Sadakane. A Space-Efficient Algorithm for the Dynamic DFS Problem in Undirected Graphs. In *In International Workshop on Algorithms and Computation*, pages 295–307, 2017. doi:10.1007/978-3-319-53925-6_23.

35 Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 950–961, 2017. doi:10.1109/FOCS.2017.92.

36 John H. Reif. Depth-First Search is Inherently Sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985. doi:10.1016/0020-0190(85)90024-9.

37 John H. Reif. A Topological Approach to Dynamic Graph Connectivity. *Inf. Process. Lett.*, 25(1):65–70, 1987. doi:10.1016/0020-0190(87)90095-0.

38 Liam Roditty. Fully Dynamic Geometric Spanners. *Algorithmica*, 62(3-4):1073–1087, 2012. doi:10.1007/s00453-011-9504-7.

39 Liam Roditty and Uri Zwick. Improved Dynamic Reachability Algorithms for Directed Graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008. doi:10.1137/060650271.

40 Liam Roditty and Uri Zwick. Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs. *SIAM J. Comput.*, 41(3):670–683, 2012. doi:10.1137/090776573.

41 Piotr Sankowski. Dynamic Transitive Closure via Dynamic Matrix Inverse (Extended Abstract). In *Symposium on Foundations of Computer Science, FOCS*, pages 509–517, 2004. doi:10.1109/FOCS.2004.25.

42 Daniel Dominic Sleator and Robert Endre Tarjan. A Data Structure for Dynamic Trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

43 Shay Solomon. Fully Dynamic Maximal Matching in Constant Update Time. In *Symposium on Foundations of Computer Science, FOCS*, pages 325–334, 2016. doi:10.1109/FOCS.2016.43.

44 Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

45 Mikkel Thorup. Fully-Dynamic Min-Cut. *Combinatorica*, 27(1):91–127, 2007. doi:10.1007/s00493-007-0045-2.