Dynamic DFS in Undirected Graphs: breaking the $O(m)$ barrier

Surender Baswana[*†] Shreejit Ray Chaudhury [*] Keerti Choudhary [*‡] Shahbaz Khan [*‡]

Given an undirected graph $G = (V, E)$ on n vertices and m edges, we address the problem of maintaining a DFS tree when the graph is undergoing *updates* (insertion and deletion of vertices or edges). We present the following results for this problem.

1. *Fault tolerant DFS tree*:
   There exists a data structure of size $\tilde{O}(m)$ [1] such that given any set $\mathcal{F}$ of failed vertices or edges, a DFS tree of the graph $G \setminus \mathcal{F}$ can be reported in $\tilde{O}(n|\mathcal{F}|)$ time.

2. *Fully dynamic DFS tree*:
   There exists a fully dynamic algorithm for maintaining a DFS tree that takes worst case $\tilde{O}(\sqrt{mn})$ time per update for any arbitrary online sequence of updates.

3. *Incremental DFS tree*:
   Given any arbitrary online sequence of edge insertions, we can maintain a DFS tree in $\tilde{O}(n)$ worst case time per edge insertion.

These are the first $o(m)$ worst case time results for maintaining a DFS tree in a dynamic environment. Moreover, our fully dynamic algorithm provides, in a seamless manner, the first deterministic algorithm with $O(1)$ query time and $o(m)$ worst case update time for the dynamic subgraph connectivity, biconnectivity, and 2-edge connectivity.

1 Introduction

Depth First Search (DFS) is a well known graph traversal technique. Right from the seminal work of Tarjan [18], DFS traversal has played the central role in the design of efficient algorithms for many fundamental graph problems, namely, biconnected components, strongly connected components, topological sorting [18], bipartite matching [12], dominators in directed graph [19], and planarity testing [13].

Let $G = (V, E)$ be an undirected connected graph on $n = |V|$ vertices and $m = |E|$ edges. DFS traversal of G starting from any vertex $r \in V$ produces a rooted spanning tree, called a DFS tree with r as its root. It takes $O(m + n)$ time to perform a DFS traversal and generate a DFS tree. Given any rooted spanning tree of graph G, all non-tree edges of the graph can be classified into two categories, namely, back edges and cross edges as follows. A non-tree edge is called a *back edge* if one of its endpoints is an ancestor of the other in the tree. Otherwise it is called a *cross edge*. A necessary and sufficient condition for any rooted spanning tree to be a DFS tree is that every non-tree edge is a back edge. Thus it can be seen that many DFS trees are possible for any given graph. However, if the traversal of the graph is performed according to the order specified by the adjacency lists of the graph, the resulting DFS tree will be unique. The ordered DFS tree problem is to compute the order in which the vertices get visited when the traversal is performed strictly according to the adjacency lists.

Most of the graph applications in real world deal with graphs that keep changing with time. These changes/updates can be in the form of insertion or deletion of vertices or edges. An algorithmic graph problem is modeled in a dynamic environment as follows. There is an online sequence of updates on the graph, and the objective is to update the solution of the problem efficiently after each update. In particular, the time taken to update the solution has to be much smaller than that of the best static algorithm for the problem. Another, and more restricted, variant of a dynamic environment is the fault tolerant environment. Here the aim is to build a compact data structure, for a given problem, that is resilient to failures of vertices/edges and can efficiently report the solution for a given set of failures. There has been a lot of work in the last two decades on dynamic and fault tolerant algorithms for various graph problems (see [1, 8]).

A dynamic graph algorithm is said to be fully dynamic if it handles both insertion as well as deletion updates. A partially dynamic algorithm is said to be incremental or decremental if it handles only insertion or only deletion updates respectively. In this paper, we address the problem of maintaining a DFS tree efficiently in any dynamic environment.

[*]Dept. of CSE, I.I.T. Kanpur, India (www.cse.iitk.ac.in), email: {sbaswana,keerti,shahbazk}@cse.iitk.ac.in, shreejit.1@gmail.com. Full version of the paper is available at http://arxiv.org/abs/1502.02481.

[†]This research was partially supported by *UGC-ISF* (the University Grants Commission of India & Israel Science Foundation) and *IMPECS* (the Indo-German Max Planck Center for Computer Science) .

[‡]This research was partially supported by Google India under the Google India PhD Fellowship Award.

[1]$\tilde{O}()$ hides the poly-logarithmic factors.

1.1 Existing results on dynamic DFS In spite of the simplicity of a DFS tree, designing any efficient parallel or dynamic algorithm for a DFS tree has turned out to be quite challenging. Reif [15] showed that the ordered DFS tree problem is a P-Complete problem. For many years, this result seemed to imply that the general DFS tree problem, that is, the computation of any DFS tree is also inherently sequential. However, Aggarwal and Anderson [2] proved that the general DFS tree problem is in *RNC* by designing a parallel randomized algorithm that takes $O(\log^3 n)$ time. Whether the general DFS tree problem is in *NC* is still a long standing open problem.

Reif [16] and later Miltersen et al. [14] proved that P-Completeness of a problem also implies hardness of the problem in the dynamic setting. The work of Miltersen et al. [14] shows that if the ordered DFS tree is updateable in $O(\mathbf{polylog}(n))$ time, then the solution of every problem in class P is updateable in $O(\mathbf{polylog}(n))$ time. In other words, maintaining the ordered DFS tree is indeed the hardest among all the problems in class P. In our view, this hardness result, which is actually for only the ordered DFS tree problem, has proved to be quite discouraging for the researchers working in the area of dynamic algorithms. This is evident from the fact that for all the static graph problems that were solved using DFS traversal in 1970's, none of their dynamic counterparts used a dynamic DFS tree.

Apart from the hardness of the ordered DFS tree problem in dynamic environment, very little progress has been achieved even for the problem of maintaining any DFS tree. Franciosa et al. [7] designed an incremental algorithm for a DFS tree in a DAG. For any arbitrary sequence of edge insertions, this algorithm takes $O(mn)$ total time to maintain a DFS tree from a given source. Recently Baswana and Choudhary [3] designed a decremental algorithm for a DFS tree in DAG that requires expected $O(mn \log n)$ total time. For undirected graphs, recently Baswana and Khan [4] designed an $O(n^2)$ total time incremental algorithm for maintaining a DFS tree. These algorithms are the only algorithms known for the dynamic DFS tree problem. Moreover, none of these existing algorithms, though designed for only a partially dynamic environment, achieves a worst case bound of $o(m)$ on the update time. So the following intriguing questions remained unanswered till date:

- Does there exist any fully dynamic algorithm for maintaining a DFS tree?
- Is it possible to achieve worst case $o(m)$ update time for maintaining a DFS tree in a dynamic environment?

Not only do we answer these open questions affirmatively for undirected graphs, we also use our dynamic algorithm for DFS tree to provide efficient solutions for a couple of well studied dynamic graph problems. Moreover, our results also handle vertex updates which are generally considered harder than edge updates.

1.2 Our results We consider a generalized notion of updates wherein an update could be either insertion/deletion of a vertex or insertion/deletion of an edge. For any set U of such updates, let $G + U$ denote the graph obtained after performing the updates U on the graph G. Our main result can be succinctly described in the following theorem.

THEOREM 1.1. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of $k \leq n$ updates, a DFS tree of $G+U$ can be reported in $O(nk \log^4 n)$ time.*

With this result at the core, we easily obtain the following results for dynamic DFS tree in an undirected graph.

1. *Fault Tolerant DFS tree*:

   Given any set of k failed vertices or edges and any vertex $v \in V$, we can report a DFS tree rooted at v for the resulting graph in $O(nk \log^4 n)$ time.

2. *Fully Dynamic DFS tree*:

   Given any arbitrary online sequence of vertex or edge updates, we can maintain a DFS tree in $O(\sqrt{mn} \log^{2.5} n)$ worst case time per update.

3. *Incremental DFS tree*:

   Given any arbitrary online sequence of edge insertions, we can maintain a DFS tree in $O(n \log^3 n)$ worst case time per edge insertion.

These are the first $o(m)$ worst case update time algorithms for maintaining a DFS tree in a dynamic environment. Recently, there has been significant work [1, 8] on establishing conditional lower bounds on the time complexity of various dynamic graph problems. A simple reduction from [1], based on the Strong Exponential Time Hypothesis (SETH), implies a conditional lower bound of $\Omega(n)$ on the update time of any fully dynamic algorithm for a DFS tree under vertex updates. We also present an unconditional lower bound of $\Omega(n)$ for maintaining a fully dynamic DFS tree explicitly under edge updates.

1.3 Applications of Fully Dynamic DFS In the static setting, a DFS tree can be easily used to answer connectivity, 2-edge connectivity and biconnectivity queries. Our fully dynamic algorithm for DFS tree thus seamlessly solves these problems for both vertex and edge updates. Further, our algorithm gives the first deterministic algorithm with $O(1)$ query time and $o(m)$ worst case update time for several well studied variants of these problems in the dynamic setting. These problems include dynamic subgraph connectivity [5, 6] and vertex update versions of dynamic biconnectivity [10, 9] and dynamic 2-edge connectivity [11]. In particular, our algorithm improves the deterministic worst case bounds for these problems, thus emphasizing the relevance of DFS trees in solving dynamic graph problems.

1.4 Main Idea Let T be a DFS tree of G. To compute a DFS tree of $G+U$ for a given set U of updates, the main idea is to make use of the original tree T itself. We preprocess the graph G using tree T to build a data structure $\mathcal{D}$. In order to achieve $o(m)$ update time, our algorithm makes use of $\mathcal{D}$ to create a *reduced* adjacency list for each vertex such that performing DFS traversal using these lists gives a DFS tree for $G+U$. In fact, these reduced adjacency lists are generated on the fly and are guaranteed to have only $\tilde{O}(n|U|)$ edges.

1.5 Outline of the paper Section 2 describes various notations that are used throughout the paper. Section 3 provides an overview of our algorithm. Our main result (Theorem 1.1)that reports a DFS tree after any set of updates in the graph is described in Section 5. In Section 6 we convert this algorithm to fully dynamic and incremental algorithms for maintaining a DFS tree using the overlapped periodic rebuilding technique. The details of the data structure $\mathcal{D}$ are described in Section 7. Due to space constraints the lower bounds and the applications of dynamic DFS have not been described in this paper. The details for the same can be found in the full version of the paper.

2 Preliminaries

Let U be any given set of updates. We add a dummy vertex r to the given graph in the beginning and connect it to all the vertices. Our algorithm starts with any arbitrary DFS tree T rooted at r in the augmented graph and it maintains a DFS tree rooted at r at each stage. It can be observed easily that each subtree rooted at any child of r is a DFS tree of a connected components of the graph $G+U$. The following notations will be used throughout the paper.

- $T(x)$: The subtree of T rooted at vertex x.
- $path(x,y)$: Path from vertex x to vertex y in T.
- $dist_T(x,y)$: The number of edges on the path from x to y in T.
- $LCA(x,y)$: The lowest common ancestor of x and y in tree T.
- $N(w)$: The adjacency list of vertex w in the graph $G+U$.
- $L(w)$: The reduced adjacency list of vertex w in the graph $G+U$.
- T^* : The DFS tree rooted at r computed by our algorithm for the graph $G+U$.
- $par(w)$: Parent of w in T^*.

A subtree T' is said to be *hanging* from a path p if the root r' of T' is a child of some vertex on the path p and r' does not belong to the path p. Unless stated otherwise, every reference to a path refers to an ancestor-descendant path defined as follows:

DEFINITION 2.1. (ANCESTOR-DESCENDANT PATH) *A path p in a DFS tree T is said to be ancestor-descendant path if its endpoints have ancestor-descendant relationship in T.*

We now state the operations supported by the data structure $\mathcal{D}$ (complete details of $\mathcal{D}$ are in Section 7). Let U below refer to a set of updates that consists of vertex and edge deletions only. For any three vertices $w,x,y \in T$, where $path(x,y)$ is an ancestor-descendant path in T the following two queries can be answered using $\mathcal{D}$ in $O(\log^3 n)$ time.

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

3 Overview

DFS traversal has the following flexibility : when the traversal reaches a vertex, say v, the next vertex to be traversed can be *any* unvisited neighbor of v. In order to compute a DFS tree for $G+U$ efficiently, our algorithm exploits this flexibility, the original DFS tree T, and the following property of DFS traversal.

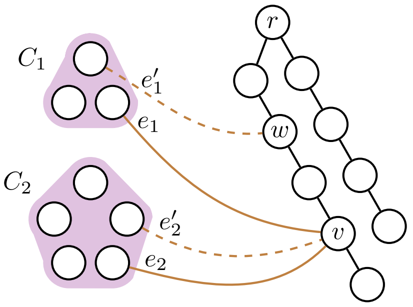

Figure 1: Edges e'_1 as well as e'_2 can be ignored during the DFS traversal.

LEMMA 3.1. (COMPONENTS PROPERTY) *Let T^* be the partially grown DFS tree and v be the vertex currently visited. Let C be any connected component in the subgraph induced by the unvisited vertices. Suppose two edges e and e' from C are incident respectively on v and some ancestor (not necessarily proper) w of v in T^*. Then it is sufficient to consider only e during the rest of the DFS traversal, i.e., the edge e' need not be scanned. (Refer to Figure 1).*

Skipping e' during the DFS traversal, as stated in the components property, is justified because e' will appear as a back edge in the resulting DFS tree. The components property can be exploited to compute reduced adjacency lists of small size for building a DFS tree as follows. Let p_0 be the path from the root to v in the partially built DFS tree T^*. From each component C of the unvisited graph, we only need to find an edge incident to the lowest vertex on p_0. Let this edge be (x, y) where $x \in p_0$ and $y \in C$. We can just add y to the reduced adjacency list $L(x)$ of x and ignore all other edges incident from C to p_0. However, as the reader may also observe, maintaining the connected components of the unvisited graph and obtaining the lowest edge incident from each of them to p_0 is a non-trivial task. Nevertheless, we are able to accomplish this task by exploiting the original DFS tree T of G as follows.

For a given set U of updates, we compute a partitioning of T into a disjoint collection of ancestor-descendant paths and subtrees such that none of these subtrees and paths contain any failed edge or vertex. An important property of this partitioning is that there are no edges from G lying between any two subtrees in $\mathcal{T}$. We refer to this partitioning as a *disjoint tree partitioning*. Note that this partitioning depends upon only the vertex and edge failures in the set U. It turns out that each component C of the unvisited graph during a DFS traversal can be represented as a union of subtrees and ancestor-descendant paths of the original DFS tree T. The lowest edges from the subtrees and the ancestor-descendant paths can be obtained by querying the data structure $\mathcal{D}$.

Let the initial disjoint tree partitioning consists of a set of ancestor-descendant paths $\mathcal{P}$ and a set of subtrees $\mathcal{T}$. The algorithm for computing a DFS tree of $G + U$ can be summarized as follows:

Perform the static DFS traversal on the graph with the elements of $\mathcal{P} \cup \mathcal{T}$ as the super *vertices. Visiting a super vertex v^* by the algorithm involves extracting an ancestor-descendant path p_0 from v^* and attaching it to the partially grown DFS tree T^*. The remaining part of v^* is added back to $\mathcal{P} \cup \mathcal{T}$ as* new *super vertices. Thereafter, the reduced adjacency list of the vertices on path p_0 is computed using the data structure $\mathcal{D}$. The algorithm then continues to find the next super vertex using the reduced adjacency lists.*

4 Disjoint Tree Partitioning

We formally define disjoint tree partitioning as follows.

DEFINITION 4.1. *Given a DFS tree T of an undirected graph G and a set U of failed vertices and edges, let A be a vertex set in $G + U$. The disjoint tree partitioning defined by A is a partition of the subgraph of T induced by A into*

1. *A set of paths $\mathcal{P}$ such that (i) each path in $\mathcal{P}$ is an ancestor-descendant path in T and does not contain any deleted edge or vertex, and (ii) $|\mathcal{P}| \le |U|$.*

2. *A set of trees $\mathcal{T}$ such that each tree $\tau \in \mathcal{T}$ is a subtree of T which does not contain any deleted edge or vertex.*

Note that for any $\tau_1, \tau_2 \in \mathcal{T}$, there is no edge between τ_1 and τ_2 because T is a DFS tree.

The disjoint tree partitioning for set $A = V \setminus \{r\}$ can be computed as follows. Let V_f and E_f respectively denote the set of failed vertices and edges associated with the updates U. We initialize $\mathcal{P} = \phi$ and $\mathcal{T} = \{T(w) \mid w \text{ is a child of } r\}$. We refine the partitioning by processing each vertex $v \in V_f$ as follows (see Figure 2 (i)).

- If v is present in some $T' \in \mathcal{T}$, we add the path from $par(v)$ to the root of T' to $\mathcal{P}$. We remove T' from $\mathcal{T}$ and add all the subtrees hanging from this path to $\mathcal{T}$.

- If v is present in some path $p \in \mathcal{P}$, we split p at v into two paths. We remove p from $\mathcal{P}$ and add these two paths to $\mathcal{P}$.

Edge deletions are handled as follows. We first remove edges from E_f that don't appear in T. Processing of the remaining edges from E_f is quite similar to the processing of V_f as described above. For each edge $e \in E_f$; just visualize deleting an imaginary vertex lying at mid-point of the edge e (see Figure 2 (ii)). It takes $O(n)$ time to process any $v \in V_f$ and $e \in E_f$.

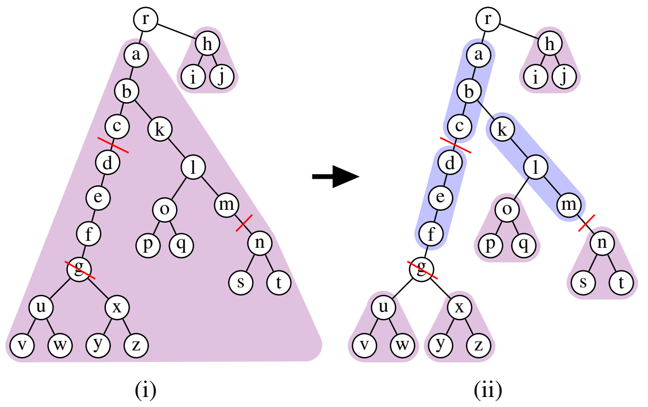

Figure 2: Disjoint tree partition for $V \setminus \{r\}$: (i) Initializing $\mathcal{T} = \{T(a), T(h)\}$ and $\mathcal{P} = \emptyset$, (ii) Final disjoint tree partition obtained after deleting vertex g and edges (c, d) and (m, n).

Note that each update can add at most one path to $\mathcal{P}$. So the size of $\mathcal{P}$ is bounded by $|U|$. The fact that T is a DFS tree of G ensures that no two subtrees in $\mathcal{T}$ will have an edge between them. So $\mathcal{P} \cup \mathcal{T}$ satisfies all the conditions stated in Definition 4.1.

LEMMA 4.1. *Given an undirected graph G with a DFS tree T and a set U of failing vertices and edges, we can find a disjoint tree partition of set $V \setminus \{r\}$ in $O(n|U|)$ time.*

5 Fault tolerant DFS Tree

We first present a fault tolerant algorithm for a DFS tree. Let U be a given set of failed vertices or edges in G. In order to compute the DFS tree T^* for $G + U$, our algorithm first constructs a disjoint tree partition $(\mathcal{T}, \mathcal{P})$ for $V \backslash \{r\}$ defined by the updates U (see Lemma 4.1). Thereafter, it can be visualized as the static DFS traversal on the graph whose (*super*) vertices are the elements of $\mathcal{P} \cup \mathcal{T}$. Note that our notion of super vertices is for the sake of understanding only.

Consider the stack-based implementation of the static algorithm for computing a DFS tree rooted at a vertex r in graph G (refer to Figure 3(i)). Our algorithm for computing DFS tree for $G + U$ (refer to Figure 3(ii)) is quite similar to the static algorithm. The only points of difference are the following.

- In the static DFS algorithm whenever a vertex is visited, it is attached to the DFS tree and pushed into the stack S. In our algorithm when a vertex u in some super vertex $v_s \in \mathcal{P} \cup \mathcal{T}$ is visited, a path starting from u is extracted from v_s and attached to the DFS tree, and this entire path is pushed into the stack S.
- Instead of scanning the entire adjacency list $N(w)$ of a vertex w, the reduced adjacency list $L(w)$ is scanned.

When a path is extracted from a super vertex v_s, the remaining unvisited part of v_s is added back to $\mathcal{T} \cup \mathcal{P}$. However, we need to ensure that the properties of disjoint tree partitioning are satisfied in the updated $\mathcal{T} \cup \mathcal{P}$. This is achieved using Procedure DFS-in-Path and Procedure DFS-in-Tree, which also build the reduced adjacency list for the vertices on the path. The construction of a sparse reduced adjacency list is inspired by Lemma 3.1 (Component property) which can be reformulated in the context of our algorithm as follows.

PROPERTY 5.1. *When a path p is attached to the partially constructed DFS tree T^* during the algorithm, for every edge (x, y), where $x \in p$ and y belongs to the unvisited graph the following condition holds. Either y is added to $L(x)$ or y' is added to $L(x')$ for some edge (x', y') where x' is a descendant (not necessarily proper) of x in p and y' is connected to y in the unvisited graph, .*

We now describe how the properties of disjoint tree partitioning and hence Property 5.1 are maintained by our algorithm when a vertex $v \in v_s$ is visited by the traversal.

1. Let $v_s = path(x, y) \in \mathcal{P}$. Exploiting the flexibility of DFS, we traverse from v to the farther end of $path(x, y)$. Now $path(x, y)$ is removed from $\mathcal{P}$ and the untraversed part of $path(x, y)$ (with length at most half of $|path(x, y)|$) is added back to $\mathcal{P}$. We refer to this as *path halving*. This technique was also used by Aggarwal and Anderson [2] in their parallel algorithm for computing DFS tree in undirected graphs. Notice that $|\mathcal{P}|$ remains unchanged or decreases by 1 after this step.
2. Let $v_s = \tau \in \mathcal{T}$. Exploiting the flexibility of a DFS traversal, we traverse the path from v to the root of τ (say x) and add it to T^*. Thereafter τ is removed from $\mathcal{T}$ and all the subtrees hanging from this path are added to $\mathcal{T}$. Observe that every newly added subtree is also a subtree of the original DFS tree T. So the properties of disjoint tree partitioning are satisfied after this step as well.

Let $path(v, x)$ be the path extracted from v_s. For each vertex w in this newly added path, we compute $L(w)$ ensuring Property 5.1 as follows.

(i) For each path $p \in \mathcal{P}$, among potentially many edges incident on w from p, we just add any one edge.

(ii) For each tree $\tau' \in \mathcal{T}$, we add at most one edge to L as follows. Among all edges incident on τ' from $path(v, x)$, if (w, z) is the edge such that w is nearest to x on $path(v, x)$, then we add z to $L(w)$. However, for the case $v_s \in \mathcal{T}$, we have to consider only the newly added subtrees in $\mathcal{T}$ for this step. This is because the disjoint tree partitioning ensures the absence of edges between v_s and any other tree in $\mathcal{T}$.

Figure 4 provides an illustration of how $\mathcal{T} \cup \mathcal{P}$ is updated when a super vertex in $\mathcal{T} \cup \mathcal{P}$ is visited.

5.1 Implementation of our Algorithm

We now describe our algorithm in full detail. Firstly we delete all the failed edges in U from the data structure $\mathcal{D}$. Now, the algorithm begins with a disjoint tree partition $(\mathcal{T}, \mathcal{P})$ which evolves as the algorithm proceeds. The state of any unvisited vertex in this partition is captured by the following three variables.
-INFO(u): this variable is set to $tree$ if u belongs to a tree in $\mathcal{T}$, and set to $path$ otherwise
-ISROOT(v): this variable is set to $True$ if v is the root of a tree in $\mathcal{T}$, and $False$ otherwise.
-PATHPARAM(v): if v belongs to some path, say $path(x, y)$, in $\mathcal{P}$, then this variable stores the pair (x, y), and $null$ otherwise.

Procedure Dynamic-DFS : For each vertex v, $status(v)$ is initially set as $unvisited$, and $L(v)$ is initialized to $\emptyset$. First a disjoint tree partition is computed for the DFS tree T based on the updates U. The procedure Dynamic-DFS then inserts the root vertex r into the stack S. Now while the stack is non-empty, the procedure repeats the following steps. It reads the top vertex from the stack. Let this vertex be w. If $L(w)$ is empty then w is popped out from the stack, else let

Procedure Static-DFS(G, r): Static algorithm to compute a DFS tree of G rooted at r.

```
Stack $S \leftarrow \emptyset$;
$Push(r)$;
$status(r) \leftarrow visited$;
while $S \neq empty$ do
    $w \leftarrow Top(S)$;
    if $N(w) = \emptyset$ then $Pop(w)$;
    else
        $u \leftarrow$ First vertex in $N(w)$;
        Remove $u$ from $N(w)$;
        if $status(u) = unvisited$ then
            $par(u) \leftarrow w$;
            $status(u) \leftarrow visited$;
            $Push(u)$;
        end
    end
end
```

(i)

Procedure Dynamic-DFS(G, U, r): Algorithm for updating the DFS tree T rooted at r for the graph $G + U$.

```
Stack $S \leftarrow \emptyset$;   $(\mathcal{T}, \mathcal{P}) \leftarrow Partition(T, U)$;
$Push(r)$;
$status(r) \leftarrow visited$;  $L(r) \leftarrow N(r)$;
while $S \neq empty$ do
    $w \leftarrow Top(S)$;  $u_0 \leftarrow w$;
    if $L(w) = \emptyset$ then $Pop(w)$;
    else
        $u \leftarrow$ First vertex in $L(w)$;
        Remove $u$ from $L(w)$;
        if $status(u) = unvisited$ then
            if INFO$(u) = tree$ then
                $\{u_1, ..., u_t\} \leftarrow$ DFS-in-Tree$(u)$;
            else if INFO$(u) = path$ then
                $\{u_1, ..., u_t\} \leftarrow$DFS-in-Path$(u)$;
            end
            for $i = 1$ to $t$ do
                $par(u_i) \leftarrow u_{i-1}$;
                $status(u_i) \leftarrow visited$;
                $Push(u_i)$;
            end
        end
    end
end
```

(ii)

Figure 3: The static (and dynamic) algorithm for computing (updating) a DFS tree. The key differences are shown in blue.

u be the first vertex in $L(w)$. If vertex u is unvisited till now, then depending upon whether $u \in \mathcal{T}$ or $u \in \mathcal{P}$, Procedure DFS-in-Tree or DFS-in-Path is executed. A path p_0 is then returned to Procedure Dynamic-DFS where for each vertex of p_0 parent is assigned and status is marked visited. The whole of this path is then pushed into stack. The procedure proceeds to the next iteration of While loop with the updated stack.

Procedure DFS-in-Tree : Let vertex u is present in tree, say $T(v)$, in $\mathcal{T}$ (the vertex v can be found easily by scanning the ancestors of u and checking their value of IsRoot). The DFS traversal enters the tree from u and leaves from the vertex v. Let $path(u, v) = \langle w_1 = u, w_2 \ldots, w_t = v \rangle$. The $path(u, v)$ is pushed into stack and attached to the partially constructed DFS tree T^*. We now update the partition $(\mathcal{P}, \mathcal{T})$ and also update the reduced adjacency list for each w_i present on $path(u, v)$ as follows.

1. For each vertex w_i and every path $path(x, y) \in \mathcal{P}$, we perform $Query(w_i, x, y)$ on the data structure $\mathcal{D}$ that returns an edge (w_i, z) such that $z \in path(x, y)$. We add z to $L(w_i)$.

2. Recall that since subtrees in T do not have any cross edge between them, therefore, there cannot be any edge incident on $path(u, v)$ from trees which are already present in $\mathcal{T}$. An edge can be incident only from the subtrees which was hanging from $path(u, v)$. $T(v)$ is removed from $\mathcal{T}$ and all the subtrees of $T(v)$ hanging from $path(u, v)$ are inserted into $\mathcal{T}$. For each such subtree, say τ, inserted into $\mathcal{T}$, we perform $Query(\tau, u, v)$ on the data structure $\mathcal{D}$ that returns an edge, say (y, z), such that $z \in \tau$ and y is nearest to u on $path(u, v)$. We insert z into $L(y)$.

Procedure DFS-in-Path : Let vertex u visited by the DFS traversal lies on a $path(v, y) \in \mathcal{P}$. Assume $dist_T(u, v) >$

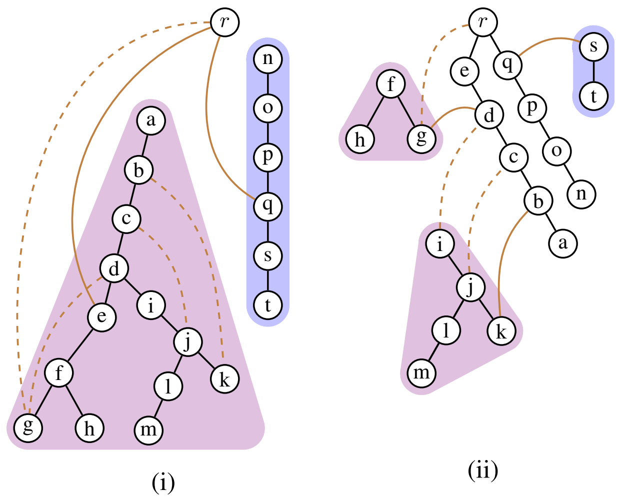

Figure 4: Visiting a super vertex from $\mathcal{T} \cup \mathcal{P}$. (i) The algorithm visits $T(a) \in \mathcal{T}$ using the edge (r, e) and the $path(n, t) \in \mathcal{P}$ using the edge (r, q). (ii) Traversal extracts $path(e, a)$ and $path(q, n)$ and augment it to T^*. The unvisited segments are added back to $\mathcal{T}$ and $\mathcal{P}$.

$dist_T(u, y)$. The DFS traversal travels from u to v (the farther end of the path). The path $path(v, y)$ in set $\mathcal{P}$ is replaced by its subpath that remains unvisited. The reduced adjacency list of each $w \in path(u, v)$ is updated in similar way as in the procedure DFS-in-Tree except that in step 2, we perform $Query(\tau, u, v)$ for each $\tau \in \mathcal{T}$.

This completes the description of the fault tolerant algorithm for DFS tree. This algorithm maintains Property 5.1 at each stage by construction given that the properties of disjoint tree partitioning are satisfied.

5.2 Correctness It can be seen that the following two invariants hold for the while loop in the Procedure Static-DFS described in Figure 3 (i). It is easy to see that these invariants imply the correctness of the algorithm, i.e., the generated tree is a rooted spanning tree where every non-tree edge is a back edge.

I_1: The sequence of vertices in the stack from bottom to top constitutes an ancestor-descendant path from r in the DFS tree computed.

I_2: For each vertex v that is popped out, all vertices in the set $N(v)$ have already been visited.

These two invariants I_1 and I_2 also hold for Procedure Dynamic-DFS described in Figure 3 (ii) as follows. Invariant I_1 holds by construction as described in our algorithm. Since our algorithm follows Property 5.1 by construction, the invariant I_2 holds (for formal proof see Lemma A.1 in Appendix). Hence our algorithm indeed computes a valid DFS tree for $G + U$.

5.3 Time complexity analysis As described earlier the disjoint tree partitioning and the components property play a key role in the efficiency of our algorithm. They allow us to limit the size of the reduced adjacency lists L, that are built during the algorithm. Our algorithm computes T^* by performing a DFS traversal on the reduced adjacency list L. Thus, the time complexity of our algorithm is $O(n + |L|)$ excluding the time required to compute L.

We first establish a bound on the size of L. In each step our algorithm extracts a path from $v_s \in \mathcal{P} \cup \mathcal{T}$ and attaches it to T^*. Let P_t and P_p denote the set of such paths that originally belonged to some tree in $\mathcal{T}$ and some path in $\mathcal{P}$, respectively. For every path $p_0 \in P_t \cup P_p$ our algorithm performs the following queries on $\mathcal{D}$.

(i) For each vertex w in p_0, we query each path in $\mathcal{P}$ for an edge incident on the vertex w. Thus the total number of edges added to L by these queries is $O(n|\mathcal{P}|)$.

(ii) If p_0 belongs to P_p, then we query for an edge from each $\tau \in \mathcal{T}$ to p_0. It follows from the path halving technique that each path in $\mathcal{P}$ reduces to at most half of its length whenever some path is extracted from it and attached to T^*. Hence the size of P_p is bounded by $|\mathcal{P}| \log n$.

(iii) If p_0 belongs to P_t, then we query for an edge from only those subtrees which were hanging from p_0. Note that these subtrees will now be added to set $\mathcal{T}$. Hence the total number of trees queried for this case will be bounded by number of trees inserted to $\mathcal{T}$. Since each subtree can be added to $\mathcal{T}$ only once, these edges are bounded by $O(n)$ throughout the algorithm.

Thus the size of L is bounded by $O\big(n(1 + |\mathcal{P}|) \log n\big)$. Since each edge added to L requires querying the data structure $\mathcal{D}$ which takes $O(\log^3 n)$ time, the total time taken to compute L is $O\big(n(1 + |\mathcal{P}| \log n) \log^3 n\big)$. Thus we have the following lemma.

LEMMA 5.1. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k failed vertices or edges (where $k \le n$), the DFS tree of $G + U$ can be reported in $O(n(1 + |\mathcal{P}| \log n) \log^3 n)$ time.*

From Definition 4.1 we have that $|\mathcal{P}|$ is bounded by $|U|$. Thus we have the following theorem.

THEOREM 5.1. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k failed vertices or edges (where $k \le n$), the DFS tree of $G + U$ can be reported in $O(nk \log^4 n)$ time.*

It can be observed that Theorem 5.1 directly implies a data structure for fault tolerant DFS tree.

5.4 Extending the algorithm to handle insertions In order to update the DFS tree, our focus has been to restrict the number of edges that are processed. For the case when the updates are deletions only, we have been able to restrict this number to $O(nk \log n)$, for a given set of k updates (failure of vertices or edges). We now describe the procedure to handle vertex and edge insertions. Let V_I be the set of the vertices inserted, and E_I be the set of edges inserted. (including the edges incident to the vertices in V_I). If there are k vertex insertions, the size of E_I is bounded by nk. So even if we add all the edges in E_I to the reduced adjacency lists, the size of L would still be bounded by $O(nk \log n)$. Hence, we perform the following two additional steps before starting the DFS traversal.

- Initialize $L(v)$ to store the edges in E_I instead of $\emptyset$. That is, $L(v) \leftarrow \{y \mid (y, v) \in E_I\}$
- Each newly inserted vertex is treated as a singleton path and added to $\mathcal{P}$. That is, $\mathcal{P} \leftarrow \mathcal{P} \cup \{x | x \in V_I\}$.

In order to establish that our algorithm, after incorporating the insertions, correctly computes a DFS tree of $G+U$, we need to ensure that all the edges *essential* for DFS traversal as described in Property 5.1 are added to L. All the essential edges from G are added to L during the algorithm itself. In case an essential edge belongs to E_I, the edge has already been added to L during its initialization. Note that the time taken by our algorithm remains unchanged since the size of L remains bounded by $O(nk \log n)$. This completes the proof of our main result stated in Theorem 1.1.

Let us consider the case when U consists of edge insertions only. In this case $\mathcal{P}$ will be an empty set. As discussed above, we initialize the reduced adjacency lists using E_I whose size is equal to $|U|$. Hence Lemma 5.1 implies the following theorem.

THEOREM 5.2. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k edge insertions (where $k \leq n$), the DFS tree of $G+U$ can be reported in $O(n \log^3 n)$ time.*

6 Fully dynamic DFS

We now describe the overlapped periodic rebuilding technique to convert our algorithm for computing a DFS tree after k updates to fully dynamic and incremental algorithms for maintaining a DFS tree. Similar technique was used by Thorup [20] for maintaining fully dynamic all pairs shortest paths.

In the fully dynamic model, we need to report the DFS tree after every update in the graph. Given the data structure $\mathcal{D}$ built using the DFS tree of the graph G, we are able to report the DFS tree of $G+U$ after $|U| = k$ updates in $\tilde{O}(nk)$ time. This becomes inefficient if k becomes large. Rebuilding $\mathcal{D}$ after every update is also inefficient as it takes $\tilde{O}(m)$ time to build $\mathcal{D}$. Thus it is better to rebuild $\mathcal{D}$ after every $|U'| = c$ updates for a carefully chosen c. Let $\mathcal{D}'$ be the data structure built using the DFS tree of the updated graph $G+U'$ with $|U'| = c$. $\mathcal{D}'$ can thus be used to process the next c updates efficiently (see Figure 5 (a)). The cost of building $\mathcal{D}'$ can thus be amortized over these c updates.

To achieve an efficient worst case update time, we divide the building of $\mathcal{D}'$ over the first c updates. This $\mathcal{D}'$ is then used by our algorithm in the next c updates, during which a new $\mathcal{D}''$ is built in a similar manner and so on (see Figure 5 (b)). The following lemma describes how this technique can be used in general for any dynamic graph problem. For notational convenience we denote any function $f(m, n)$ as f.

LEMMA 6.1. *Let 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 D can be build in $O(f)$ time and the solution for graph $G+U$ can be reported in $O(h+|U| \times g)$ time, then D can be used to report the solution after every update in worst case $O(\sqrt{fg}+h)$ update time, given that $\sqrt{f/g} \leq n$.*

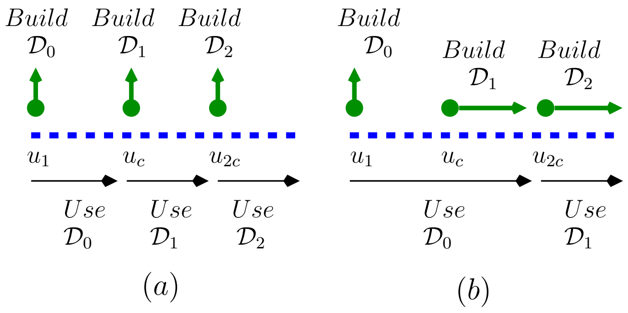

Figure 5: (a) Fully dynamic algorithm with amortized update time. (b) De-amortization of the algorithm.

Proof. We first present an algorithm that achieves amortized $O(\sqrt{fg}+h)$ update time. It is based on the simple idea of periodic rebuilding. Given the input graph G_0 we preprocess it to compute the data structure D_0 over it. Now let $u_1, ..., u_c$ $(c \leq n)$ be the sequence of first c updates on G_0. To report the solution after i^{th} update we use D_0 to compute the solution for $G_0 + \{u_1, ..., u_i\}$. This takes $O(h+(i \times g))$ time. So the total time for preprocessing and handling the first c updates is $O(f + \sum_{i=1}^{c} h + (i \times g))$. Therefore, the average time for the first c updates is $O(f/c + c \times g + h)$. Minimizing this quantity over c gives the optimal value $c_0 = \sqrt{f/g}$ which is bounded by n. So, after every c_0 updates we rebuild our data structure and use it for the next c_0 updates (see Figure 5(a)). Substituting the value of c_0 gives the amortized time complexity as $O(\sqrt{fg} + h\ )$.

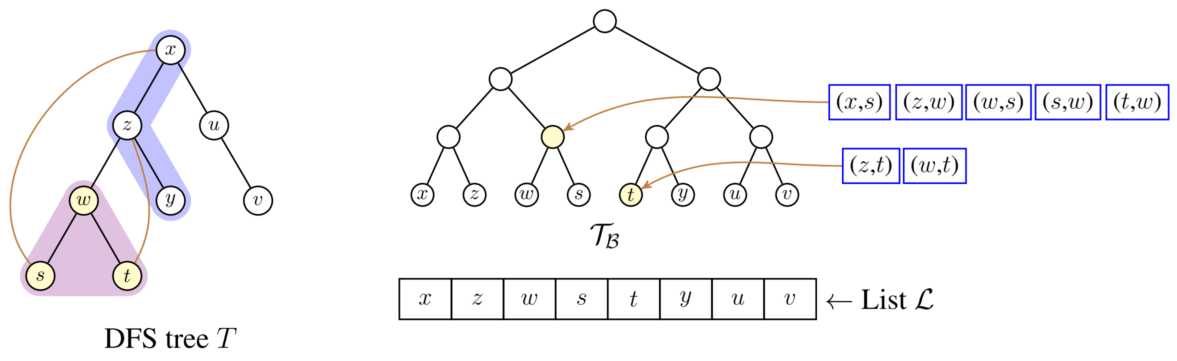

Figure 6: (i) The highest edge from subtree $T(w)$ on $path(x, y)$ is edge (x, s) and the lowest edges are edge (z, w) and (z, t). (ii) The vertices of $T(w)$ are represented as union of two subtrees in segment tree $\mathcal{T}_\mathcal{B}$.

The above algorithm can be de-amortized as follows. Let $G_1, G_2, G_3, \ldots$ be the sequence of graphs obtained after $c_0, 2c_0, 3c_0, ..$ updates. We use the data structure D_0 built during preprocessing to handle the first $2c_0$ updates. Also after the first c_0 updates we start building the data structure D_1 over G_1. This D_1 is built in c_0 steps, thus the extra time spent per update is $f/c_0 = O(\sqrt{fg})$ only. We use D_1 to handle the next c_0 updates on graph G_2, and also in parallel compute the data structure D_2 over the graph G_2. (See Figure 5(b)). Since the time for building each data structure is now divided in c_0 steps, we have that the worst case update time as $O(\sqrt{fg} + h\ )$.

The above lemma combined with Theorems 1.1 and 5.2 directly implies the following results for the fully dynamic DFS tree problem and the incremental DFS tree problem, respectively.

(For the theorem below we use $f = m \log n$, $g = n \log^4 n$ and $h = 0$.)

THEOREM 6.1. *There exists a fully dynamic algorithm for maintaining a DFS tree in an undirected graph that uses $O(m \log n)$ preprocessing time and can report a DFS tree after each update in the worst case $O(\sqrt{mn} \log^{2.5} n)$ time. An update in the graph can be insertion / deletion of an edge as well as a vertex.*

(For the theorem below we use $f = m \log n$, $g = \log n$ and $h = n \log^3 n$, since $O(n \log^3 n) = O(k \times g + h)$ for $k \leq n$).

THEOREM 6.2. *There exists an incremental algorithm for maintaining a DFS tree in an undirected graph that uses $O(m \log n)$ preprocessing time and can report a DFS tree after each edge insertion in the worst case $O(n \log^3 n)$ time.*

7 Data Structure

The efficiency of our algorithm relies heavily on the data structure $\mathcal{D}$. Its construction employs a combination of two well known techniques, namely, heavy-light decomposition [17] and suitable augmentation of a binary tree (segment tree) as follows.

1. Perform a pre-order traversal of the tree T with the following restriction: Upon visiting a vertex $v \in T$, the child of v that is visited first is the one storing the largest subtree. Let $\mathcal{L}$ be the list of vertices ordered by this traversal.

2. Build a segment tree $\mathcal{T}_\mathcal{B}$ whose leaf nodes from left to right represent the vertices in list $\mathcal{L}$.

3. We augment each node z of $\mathcal{T}_\mathcal{B}$ by a binary search tree $\mathcal{E}(z)$ storing all the edges $(u, v) \in E$ where u is a leaf node in the subtree rooted at z in $\mathcal{T}_\mathcal{B}$. These edges are sorted according to the position of the second endpoint in $\mathcal{L}$.

The construction of $\mathcal{D}$ described above ensures the following properties which are helpful in answering a query $Query(T(w), x, y)$ (see Figure 6).

- $T(w)$ is present as an interval of vertices in $\mathcal{L}$ (by step 1). Moreover, this interval can be expressed as a union of $O(\log n)$ disjoint subtrees in $\mathcal{T}_\mathcal{B}$ (by step 2). Let these subtrees be $T_1, \ldots, T_q$.

- It follows from the heavy-light decomposition used in step 1 that path $path(x, y)$ can be divided into $O(\log n)$ subpaths $path(x_1, y_1), \ldots, path(x_\ell, y_\ell)$ such that each subpath $path(x_i, y_i)$ is an interval in $\mathcal{L}$.

- Let z_j be the root of subtree T_j in $\mathcal{T}_\mathcal{B}$. Then it follows from step 3 that any query $Query(T_j, x_i, y_i)$ can be answered by a single predecessor or successor query on the BST $\mathcal{E}(z_j)$ in $O(\log n)$ time.

To answer $Query(T(w), x, y)$, we thus find the edge closest to x among all the edges reported by the queries

$\{Query(T_j, x_i, y_i) | 1 \leq j \leq q$ and $1 \leq i \leq \ell\}$. Thus $Query(T(w), x, y)$ can be answered in $O(\log^3 n)$ time. Notice that $Query(w, x, y)$ can be considered as a special case where $q = 1$ and T_1 is the leaf node of $\mathcal{T_B}$ representing w. The space required by $\mathcal{D}$ is $O(m \log n)$ as each edge is stored at $O(\log n)$ levels in $\mathcal{T_B}$. Now, the segment tree $\mathcal{T_B}$ can be built in linear time. Further, for every node u, the sorted list of edges in $\mathcal{E}(u)$ can be computed in linear time by merging the sorted lists of its children. Thus the binary search tree $\mathcal{E}(u)$ for each node $u \in \mathcal{T_B}$ can be built in time linear in the number of edges in $\mathcal{E}(u)$. Hence the total time required to build this data structure is $O(m \log n)$. Thus we have the following theorem.

THEOREM 7.1. *The queries $Query(T(w), x, y)$, $Query(w, x, y)$ on T can be answered in $O(\log^3 n)$ worst case time using a data structure $\mathcal{D}$ of size $O(m \log n)$, which can be build in $O(m \log n)$ time.*

Note: Our algorithm also requires deletion of edges from $\mathcal{D}$. An edge can be deleted from $\mathcal{D}$ by deleting the edge from the binary search tree stored at its end points and their ancestors in $\mathcal{T_B}$. Since a deletion in binary search tree takes $O(\log n)$ time, an edge can be deleted from $\mathcal{D}$ in $O(\log^2 n)$ time.

References

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443, 2014.

[2] Alok Aggarwal and Richard J. Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988.

[3] Surender Baswana and Keerti Choudhary. On dynamic DFS tree in directed graphs. In *MFCS, Proceedings, Part II*, pages 102–114, 2015.

[4] Surender Baswana and Shahbaz Khan. Incremental algorithm for maintaining DFS tree for undirected graphs. In *ICALP, Proceedings, Part I*, pages 138–149, 2014.

[5] Timothy M. Chan, Mihai Patrascu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. In *FOCS*, pages 95–104, 2008.

[6] Ran Duan. New data structures for subgraph connectivity. In *ICALP, Proceedings, Part I*, pages 201–212, 2010.

[7] 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.

[8] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30, 2015.

[9] Monika Rauch Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, 1995.

[10] Monika Rauch Henzinger. Improved data structures for fully dynamic biconnectivity. *SIAM J. Comput.*, 29(6):1761–1815, 2000.

[11] 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.

[12] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[13] John E. Hopcroft and Robert Endre Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[14] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994.

[15] John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.

[16] John H. Reif. A topological approach to dynamic graph connectivity. *Inf. Process. Lett.*, 25(1):65–70, 1987.

[17] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[18] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[19] Robert Endre Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, 1974.

[20] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *STOC*, pages 112–119, 2005.

A Appendix

A.1 Proof of I_2 using Property 5.1

LEMMA A.1. *If Property 5.1 is maintained by the procedure Dynamic-DFS, then invariant I_2 will hold true at each stage of the algorithm.*

Proof. We give a proof by contradiction as follows. Assume that x is the first vertex that is popped out of the stack before some vertex $y \in N(x)$ is visited. Consider the time when a path p containing x was pushed in the stack. Clearly $y \notin L(x)$, hence using Property 5.1 we know that some $y' \in L(x')$ is connected to y in the unvisited graph where x' is a descendant (not necessarily proper) of x in p. Let p^* be a path between y' and y in the unvisited graph.

Now consider the time when x is popped out of the stack. Clearly all its descendants have been popped out, so y' has been visited by the traversal. Thus p^* can be divided into two non-empty sets A and B denoting visited and unvisited vertices of p^* respectively. Here $y' \in A$ and $y \in B$, thus clearly for some vertex in A invariant I_2 is not satisfied. This contradicts our assumption that x is the first vertex that is popped out of the stack for which I_2 is not satisfied. Thus maintenance of Property 5.1 ensures the invariant I_2 in our algorithm.