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

Surender Baswana∗ Shreejit Ray Chaudhury Keerti Choudhary†
Shahbaz Khan‡

Department of Computer Science and Engineering,
I.I.T. Kanpur, Kanpur - 208016, India.

Depth first search (DFS) tree is a fundamental data structure for solving various problems in graphs. It is well known that it takes $O(m + n)$ time to build a DFS tree for a given 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 or 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 $O(m \log n)$ such that given any $k$ updates, a DFS tree for the resulting graph can be reported in $\tilde{O}(nk)$ worst case time. When the $k$ updates are deletions only, this directly implies an $\tilde{O}(nk)$ time fault tolerant algorithm for a DFS tree.

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.

Our results are the first $o(m)$ worst case time results for maintaining a DFS tree in a dynamic setting. Our fully dynamic algorithm provides, in a seamless manner, the first non-trivial algorithm with $O(1)$ query time and $o(m)$ worst case update time for the dynamic subgraph connectivity, bi-connectivity, 2 edge-connectivity. We also present the conditional lower bound of $\Omega(n)$ time per update for maintenance of a DFS tree.

∗This research was partially supported by the Indo-German Max Planck Center for Computer Science (IMPECS)
†This research was partially supported by Google India under the Google India PhD Fellowship Award.
‡This research was partially supported by Google India under the Google India PhD Fellowship Award.
1$\tilde{O}(\cdot)$ hides the poly-logarithmic factors.
1 Introduction

Depth First Search (DFS) is a well known graph traversal technique. Tarjan, in his seminal work [28], demonstrated the power of DFS traversal by presenting efficient algorithms for graph problems as bi-connected components and strongly connected components. Later on DFS traversal was used to solve various fundamental graph problems as topological sorting [30], bipartite matching [18], dominators in directed graph [29], planarity testing [19], edge and vertex connectivity [13] etc.

DFS traversal is a recursive algorithm to traverse a graph. This traversal produces a rooted spanning tree (or forest in case the graph is not connected), called DFS tree (forest). Let \( G = (V, E) \) be an undirected graph on \( n = |V| \) vertices and \( m = |E| \) edges. It takes \( O(m + n) \) time to perform a DFS traversal and generate its DFS tree (forest).

Given any rooted spanning tree of graph \( G \), all non-tree edges of the graph can be partitioned into two categories, namely, back edges and cross edges as follows. A non-tree edges is called a back edge if one of its end vertex 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 all its non-tree edges are back edges. So many DFS trees are possible and for all algorithmic applications mentioned above, any DFS tree works equally well. However, if the traversal of the graph is performed strictly according to the order specified by the adjacency lists of the graph, then the resulting DFS tree is unique. The ordered DFS problem is to compute the order in which the vertices get visited when the traversal is performed strictly according to the adjacency list.

Most of the graph applications in real world deal with graphs that keep changing with time. These changes can be in the form of insertion or deletion of vertices or edges. An algorithmic graph problem is modeled in the dynamic environment as an online sequence of insertion and deletion of vertices or edges. The aim is to maintain the solution of the given problem after each update using some clever data structure such that the time taken to update the solution after any edge update is much smaller than that of the best static algorithm. Another, and more restricted, variant of a dynamic graph environment is the fault tolerant environment. Here the aim is to build a compact data structure that is resilient to failures of vertices/edges, and can efficiently report the solution for a given set of failures. In the last two decades, many elegant dynamic algorithms have been designed for various graph problems such as connectivity [16, 17, 20, 22], reachability [25], shortest path [9, 26], spanners [4], and min-cut [31].

In this article, we address the problem of maintaining a DFS tree efficiently in a dynamic environment.

1.1 Existing results on dynamic DFS

In spite of the simplicity and elegance of a DFS tree, its parallel and dynamic versions have turned out to be quite challenging. Reif [23] showed that ordered DFS is a P-Complete problem. For quite some time, this result seemed to imply that general DFS is also inherently sequential. However, Aggarwal and Anderson [2] proved that general DFS is in RNC by designing a parallel randomized algorithm for DFS that takes \( O(\log^3 n) \) time. Whether general DFS problem is in NC is still a long standing open problem.

Reif [24] and later Milterson et al. [21] proved that P-Completeness of a problem also implies hardness of the problem in the dynamic setting. The work of Milterson et al. [21] implies that if ordered DFS tree is updatable in \( O(\text{polylog}(n)) \) time, then solution of every problem in class \( P \) is updatable in \( O(\text{polylog}(n)) \) time. In other words, maintaining the ordered DFS tree dynamically is indeed the hardest among all the problems in class \( P \). In our view, the hardness result of the ordered DFS 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 static problems for which DFS traversal provided an elegant solution in 1970’s, the dynamic counterparts did not employ any dynamic DFS tree approach [16, 17, 20, 25, 6, 7, 10].

Very little progress has been made in the last 30 years on dynamic DFS tree problem. Franciosa et al. [14] designed an algorithm for maintaining DFS tree in a DAG under insertion of edges. For any
arbitrary sequence of edge insertions, this algorithm takes $O(mn)$ total time to maintain the DFS tree from a given source. For undirected graphs, recently Baswana and Khan [3] designed an $O(n^2)$ time incremental algorithm for maintaining a DFS tree. These incremental algorithms are the only non-trivial results known for the dynamic DFS tree problem. None of these existing algorithms, though designed for only partially dynamic environment, achieve a worst case bound of $o(m)$ on the update time. So till date, the following intriguing questions still remained unanswered:

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

### 1.2 Our results

Not only we answer these two questions affirmatively for undirected graphs, we also use our dynamic algorithm of DFS tree for providing efficient solutions for a couple of well studied dynamic graph problems.

We consider an extended 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 from set $U$ 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 $k$ updates (where $k \leq n$), the DFS tree of $G + U$ can be reported in $O(nk \log^4 n)$ time.

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

1. **Fault Tolerant DFS tree**:
   
   Given any set $\mathcal{F}$ of $k$ failing vertices or edges and any vertex $v \in V$, we can report a DFS tree rooted at $v$ in $O(nk \log^4 n)$ time.

2. **Fully Dynamic DFS tree**:
   
   Given any arbitrary 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 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 $o(m)$ update time algorithms for the maintaining a DFS tree in a dynamic setting.

We also present conditional lower bounds of $\Omega(n)$ on the update time for maintaining DFS tree in a fully dynamic environment. Therefore, our results are off from optimal by a factor of $\tilde{O}(k)$ for the fault tolerant setting and by a factor of $\tilde{O}(\sqrt{m/n})$ for the fully dynamic maintenance of DFS tree. We now demonstrate the applications of our fully dynamic DFS tree algorithm

**Dynamic subgraph connectivity, bi-connectivity, and 2 edge-connectivity**

The dynamic subgraph connectivity problem is defined as follows. Given an undirected graph, the status of any vertex can be switched between *active* and *inactive* in an update. The goal is to efficiently answer any online connectivity query on the subgraph induced by the active vertices. This problem can be solved...
using dynamic connectivity data structures [12, 15, 17] that answers connectivity queries under an online sequence of edge updates. This is because switching the state of a vertex is equivalent to \( O(n) \) edge updates. Chan [6] introduced this problem and showed that it can be solved more efficiently. He gave an algorithm using FMM (fast matrix multiplication) that achieves \( O(m^{0.94}) \) amortized update time and \( \tilde{O}(m^{1/3}) \) query time. Later Chan et al. [7] presented a new algorithm that improves the amortized update time to \( \tilde{O}(m^{2/3}) \). Further they established a lower bound of \( \Omega(\sqrt{m}) \) time per update. They also mentioned the following among the open problems.

1. Is it possible to achieve constant query time with worst case sublinear update time?

2. Can non trivial updates be obtained for richer queries such as counting the number of connected components?

Duan [10] answered the first question in affirmative but at the expense of much higher update time. We answer both these questions affirmatively. Our fully dynamic algorithm directly provides an \( \tilde{O}(\sqrt{mn}) \) update time and \( O(1) \) query time algorithm for the dynamic subgraph connectivity problem. Our algorithm maintains the number of connected components simply as a byproduct. In fact, our fully dynamic algorithm for DFS tree solves a generalization of dynamic subgraph connectivity - in addition to just switching the status of vertices, it allows insertion of new vertices as well.

Exploiting the rich structure of DFS tree, we also obtain \( \tilde{O}(\sqrt{mn}) \) update time algorithms for dynamic subgraph bi-connectivity and dynamic subgraph 2 edge-connectivity in a seamless manner. Figure 1 illustrates our results and the existing results in the right perspective. A variant of dynamic subgraph connectivity problem has been studied by Patrascu and Thorup [22]. For any \( k \) edge failures, their data structure takes \( \tilde{O}(k) \) update time such that any subsequent connectivity query can be answered in \( O(\log \log n) \) time. Later Duan and Pettie [11] presented a data structure for vertex failures. For any \( k \) vertex failures, their data structure takes \( \tilde{O}(k^{6}) \) update time and answers any subsequent connectivity query in \( O(k) \) time.

Unlike all the previous algorithms for dynamic subgraph connectivity, which use heavy machinery of existing dynamic algorithms, our algorithms are arguably much simpler and self contained.

| References          | Update Time        | Query Time      | Space          |
|---------------------|--------------------|-----------------|----------------|
| Frederickson [15]   | \( O(n\sqrt{m}) \) | \( O(1) \)     | \( O(m) \)     |
| Eppstein et. al. [12] (1997) | \( \tilde{O}(n) \) amortized | \( \tilde{O}(1) \) | \( O(m) \)     |
| Holm et al. [17]   | \( \tilde{O}(m^{0.94}) \) amortized | \( \tilde{O}(m^{1/3}) \) | \( \tilde{O}(m) \) |
| Chan [6]           | \( \tilde{O}(m^{2/3}) \) amortized | \( \tilde{O}(m^{1/3}) \) | \( \tilde{O}(m^{4/3}) \) |
| Chan et al. [7]    | \( \tilde{O}(m^{4/5}) \) amortized | \( \tilde{O}(m^{1/5}) \) | \( \tilde{O}(m) \) |
| Duan [10] (2010)   | \( \tilde{O}(\sqrt{mn}) \) | \( O(1) \)     | \( \tilde{O}(m) \) |
| Duan [10] (2010)   | \( \tilde{O}(\sqrt{mn}) \) | \( O(1) \)     | \( \tilde{O}(m) \) |
| New                | \( \tilde{O}(\sqrt{mn}) \) | \( O(1) \)     | \( \tilde{O}(m) \) |

Figure 1: Current-state-of-the-art of the algorithms for the dynamic subgraph connectivity. (*) indicates that the result also solves subgraph bi-connectivity and 2-edge connectivity.

2 Preliminaries

Let \( G = (V, E) \) be any given undirected graph \( G = (V, E) \) on \( n = |V| \) vertices and \( m = |E| \) edges. 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 component 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.
- distT(x, y): The length of 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 neighbors of vertex w in graph G + U.
- T*: The DFS tree rooted at r computed by our algorithm for the graph G + U.
- L(w): The subset of neighbors of w in graph G + U used to build T* by our algorithm.
- par(w): Parent of w in T*.

We keep a data structure D (refer to Appendix A.4) defined for the tree T and updates U. For any three vertices w, x, y ∈ T, the following two queries can be answered using D in O(log^3 n) time.

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

3 Overview

Static DFS traversal has to explore each edge and thus needs Θ(m + n) time. In order to compute the DFS tree for G + U in o(m) time, the aim is to explore only a small subset of neighbors L(v) ⊂ N(v) of every vertex v. For this purpose, we exploit the flexibility inherent in DFS traversal and an important property. The flexibility of DFS is that once traversal reached a vertex v, the next vertex to be traversed can be any one among its unvisited neighbors of v. We now state the important property of DFS traversal.

**Lemma 3.1 (Components Property)** While DFS traversal reaches a vertex v, let the graph induced by all the unvisited vertices consists of connected components C1, ..., Ct. Suppose any component Cℓ has edges e and e' incident respectively on v and some (not necessarily proper) ancestor w of v in the partially grown DFS tree. If we traverse e, then e' will always be a back edge and so won’t appear in the DFS tree. Therefore, we just need to keep only e in L(v) and skip e'.

In order to exploit the flexibility and component property of DFS traversal, the DFS tree T for the original graph G turns out to be very useful. We partition the DFS tree T into a disjoint collection of paths P and subtrees T such that P ⊆ |U|. While computing the DFS tree of G + U, our algorithm tries to preserve T to minimize the computation. The entire algorithm can be seen as the static DFS algorithm executed on the graph with elements of P ∪ T as its super vertices wherein we grow L(v) for vertices lazily and cleverly. It performs one of the following two steps depending upon whether the next super vertex traversed is a subtree or a path in the partition P ∪ T.

1. If the next super vertex is a path p(x, y) ∈ P, let v ∈ p be the vertex which is visited first. Exploiting flexibility of DFS, we traverse from v to x and thus add (hence preserve) path p(v, x) to T*. The untraversed part of p(x, y) is added to the P. For each vertex w of the newly added path p(v, x), we compute L(w) by exploiting components property as follows.
For each tree $\tau \in \mathcal{T}$, among all edges incident on $\tau$ from path$(v, x)$, let $(w, z)$ be the edge such that $w \in$ path$(v, x)$ is nearest to $x$ on path$(v, x)$. We insert edge $(w, z)$ to $L(w)$.

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

In order to perform these two steps, we employ the data structure $\mathcal{D}$. The efficiency of our dynamic algorithm relies heavily on this data structure. Its construction employs a novel combination of two well known techniques, namely, heavy-light decomposition and suitably augmented segment tree.

2. If the next super vertex is a subtree $\tau \in \mathcal{T}$, let $v \in \tau$ be the vertex which is visited first. Then, exploiting flexibility of DFS, we traverse the path from $v$ to the root of $\tau$ and add it to $T^*$. All the subtrees hanging from this path get added to the pool of $\mathcal{T}$. The components property is now used on this preserved path in the same way as in the previous case.

Figure 2 provides an illustration of this clever DFS traversal on the super vertices of $\mathcal{P} \cup \mathcal{T}$. Note that our notion of super vertices is for the sake of understanding only. An element of $\mathcal{P} \cup \mathcal{T}$ representing a super vertex may disintegrate during the algorithm as evident from Figure 2.

While building the DFS tree for $G + U$ as sketched above, the partition $\mathcal{P} \cup \mathcal{T}$ evolves as the algorithm proceeds. Note that the number of paths in $\mathcal{P}$ remains bounded by $|U|$, though a single path may break and appear multiple times. The number of edges added to $L(v)$, $\forall v \in V$ during the algorithm remain bounded by $O(nk^*)$, where $k^*$ is the number of paths added to $\mathcal{P}$ during the entire algorithm. Notice that $k^*$ can be much larger than $|U|$ because the DFS traversal may enter a path many times during the algorithm. In order to limit $k^*$ we use the technique of path halving: whenever the DFS entries a path $p \in \mathcal{P}$, we traverse to the farther end of the path so that the length of the untraversed portion of $p$, to be placed back into $\mathcal{P}$, is always at most half of $p$. This limits the value of $k^*$ to $O(|U| \log n)$. Similar idea was also used by Aggarwal and Anderson [2] in the parallel algorithm for computing DFS tree in undirected graphs.

4 Disjoint Tree Partitioning of DFS tree

We begin with the definition for an ancestor-descendant path.
**Definition 4.1 (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 \( T \).

The disjoint tree partitioning of \( T \) can be defined as follows.

**Definition 4.2 (Disjoint tree partition)** Given a set of updates \( U \), a disjoint tree partition divides \( T \) into

1. A set of paths \( P \) satisfying that (i) each path in \( P \) is an ancestor-descendant path in \( T \) which is intact in \( G + U \), and (ii) \(|P| \leq |U|\).
2. A set of trees \( T \) such that each tree \( \tau \in T \) is a subtree of \( T \), and is intact in \( G + U \).

This disjoint tree partition of \( T \) can be computed by only looking at the deletions in set \( U \). Let \( V_D \) and \( E_D \) respectively denote the set of vertices and edges that are deleted. We initialize \( P = \emptyset \) and \( T = \{ T(w) \mid w \text{ is a child of } r \} \). We process each vertex \( v \in V_D \) as follows (see Figure 3).

- If \( v \) is present in some \( T' \in T \), we add the path from \( \text{par}(v) \) to the root of \( T' \) to \( P \). All subtrees hanging from children of \( v \) as well as all subtrees hanging from this path are added to \( T \).
- If \( v \) is present in some path \( p \in P \), we split \( p \) at \( v \) into two paths. We remove \( p \) from \( P \) and add these two paths to \( P \).

We first remove edges from \( E_D \) that don’t appear in \( T \). Processing of the remaining edges from \( E_D \) is quite similar to the processing of \( V_D \) as described above: For each edge \( e \in E_D \), just visualize deleting an imaginary vertex lying at mid-point of the edge \( e \).

Now each update can add at most one path to \( P \) limiting size of \( P \) to at most \(|U|\). The fact that \( T \) is a DFS tree of \( G \) ensures that no two subtrees in \( T \) will have an edge between them. So \( P \cup T \) satisfies all the properties stated in Definition \( 4.2 \). It can also be seen easily that this partitioning can be computed in \( O(n) \) time if we process the updates in a top-down fashion in \( T \).

**Lemma 4.1** Given an undirected graph \( G \) with a DFS tree \( T \) and a set \( U \) of \( k \) updates on \( G \), we can find a disjoint tree partition of \( T \) in \( O(n) \) time.
5 Algorithm

Consider the static algorithm for computing a DFS tree rooted at a vertex \( r \) in graph \( G \). Figure 4(i) presents a pseudo code of a stack-based implementation of this algorithm. The following two invariants, ensure the correctness of this algorithm, that is, every non-tree edge will be a back edge.

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

\[ I_2: \text{For each vertex } v \text{ that is popped out, all vertices in the set } N(v) \text{ have already been visited.} \]

We first present our dynamic algorithm for the case when the updates are restricted to deletions only. Let \( U \) be the set of updates on \( G \). Our dynamic algorithm updates \( T \) by first computing a disjoint tree partition \((T,P)\) for \( T \) based on 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 \( P \cup T \). Refer to Figure 4(ii) for the pseudo code of this algorithm and notice a huge similarity with the static algorithm on the left. The points of difference are the following.

- Whenever a vertex \( u \) in some super vertex \( v_s \in P \cup T \) is visited for the first time, a path starting from \( u \) is extracted from \( v_s \). Instead of just pushing \( u \) in the stack \( S \), this entire path is pushed in \( S \), and also attached to the partially constructed DFS tree. This is done by Procedure DFS-in-Tree if \( v_s \in T \) and by Procedure DFS-in-Path if \( v_s \in P \).

- Instead of scanning the entire adjacency list \( N(w) \) of a vertex \( w \), a reduced adjacency list \( L(w) \) is scanned. This list \( L(w) \) is constructed lazily during the algorithm.

Our algorithm ensures that the total size of the reduced adjacency lists \( L(w), \forall w \in V \) will be \( O(nk \log n) \). It is interesting to note here that by processing even this sparse subset of edges, the algorithm produces a valid DFS tree for \( G + U \). The construction of a sparse \( L(w) \) is inspired by Lemma 3.1 (Component property) which can be described in the context of our algorithm as follows.

**Property 5.1** Each time a path \( p \) is attached to partially constructed DFS tree we have that for each connected component \( C \) of the unvisited graph, if \( w \) is the lowest vertex on \( p \) adjacent to \( C \) then at least one vertex from \( N(w) \cap C \) is added to set \( L(w) \). (By lowest we mean a vertex on \( p \) at maximum depth in the partially constructed DFS tree).

The two invariants \( I_1 \) and \( I_2 \) also hold for Procedure Dynamic-DFS as follows. Invariant \( I_1 \) holds by construction as described above. Invariant \( I_2 \) can be shown to hold using Property 5.1 which is ensured by our algorithm (see Lemma A.1 in Appendix). Hence our algorithm indeed computes a valid DFS tree for \( G + U \).

We now describe our algorithm in full detail. The algorithm begins with a disjoint tree partition \((T,P)\) which evolves as the algorithm proceeds. The state of any unvisited vertex in this partition is captured by the following three variables.

- \( \text{-INFO}(u) \): this variable is set to \textit{tree} if \( u \) belongs to a tree in \( T \), and set to \textit{path} otherwise.
- \( \text{-ISROOT}(v) \): this variable is set to \textit{True} if \( v \) is the root of a tree in \( T \), and \textit{False} otherwise.
- \( \text{-PARAM}(v) \): if \( v \) belongs to some path, say path \((x,y)\), in \( P \), then this variable stores the pair \((x,y)\), and \textit{null} otherwise.

**Procedure Dynamic-DFS**: For each vertex \( v \), \textit{status}(\( v \)) is initially set as \textit{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
Procedure Static-DFS($G, r$): Static algorithm to compute a DFS tree of $G$ rooted at $r$.

1. Stack $S \leftarrow \emptyset$;
2. $Push(r)$;
3. $status(r) \leftarrow visited$;
4. while $S \neq empty$ do
   5. $u \leftarrow Top(S)$;
   6. if $N(u) = \emptyset$ then $Pop(w)$ else
   7. Remove $u$ from $N(w)$;
   8. if $status(u) = unvisited$ then
      9. $par(u) \leftarrow w$;
      10. $Push(u)$;
   11. end
   12. end

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

1. Stack $S \leftarrow \emptyset$; $(T, P) \leftarrow Partition(T, U);$ $Push(r);$ $status(r) \leftarrow visited; L(r) \leftarrow N(r);$ $while S \neq empty do$
2. $w \leftarrow Top(S);$ $if L(w) = \emptyset$ then $Pop(w)$ else
3. $u \leftarrow First vertex in L(w);$ Remove $u$ from $L(w);$ $if status(u) = unvisited then$
4. $par(u) \leftarrow w;$ $if INFO(u) = tree then$
5. $DFS-in-Tree(u);$ $else$
6. $(v, d) \leftarrow PATHPARAM(u);$ $DFS-in-Path(u, v, d);$ $end$
7. $end$
8. end

Figure 4: The static (and dynamic) algorithm for computing (updating) a DFS tree.

empty then $w$ is popped out from the stack, else let $u$ be the first vertex in $L(w)$. If vertex $u$ is unvisited till now, then the DFS traversal visits $u$ and assigns $w$ as its parent. Thereafter, the procedure DFS-in-Path or DFS-in-Tree is executed depending upon whether $u$ belongs to a path or tree in $T \cup P$. Each of these procedures basically performs the following tasks: (i) extracting a path $p_0$ from $T \cup P$, (ii) computing $L(v)$ for each $v \in p_0$, (iii) attaching $p_0$ to $T^*$, (iv) pushing vertices of $p_0$ in the 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 $T$ (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 $path(u, v)$ is pushed into stack and attached to the partially constructed DFS tree $T^*$. We now update the partition $(P, T)$ 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 $path(x, y) \in P$, we perform $Query(w_i, x, y)$ on the data structure $D$ that returns an edge $(w_i, z)$ such that $z \in path(x, y)$. We add $z$ to $L(w_i)$.
   Note: There can be other neighbors of $w_i$ also lying on $path(x, y)$ but Property 5.1 allows us to ignore all of them.

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 $T$. An edge can be incident only from the subtrees which was hanging from $path(u, v)$. $T(v)$ is removed from $T$ and all the subtrees of $T(v)$ hanging from $path(u, v)$ are inserted into $T$. For each such subtree, say $\tau$, 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 $L(y)$.

Note: There can be multiple edges incident on $path(v, u)$ from tree $\tau$. But using Property 5.1, it is sufficient to add the only edge $(y, z)$ to the reduced adjacency list.

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

The reader may refer to Appendix A.2 for pseudo code of Procedures DFS-in-Tree and DFS-in-Path. This completes the description of the dynamic algorithm for DFS tree. This algorithm maintains Property 5.1 at each stage. For a formal proof, kindly refer to Appendix A.3

**Time complexity analysis**

We now analyze the time complexity of our algorithm. Let $P_t$ denote the set of paths attached to $T^*$ that originally belonged to some tree in $\mathcal{T}$. Similarly, let $P_p$ denote the set of paths attached to $T^*$ that originally belonged to some path in $\mathcal{P}$. For any path $p_0 \in P_t \cup P_p$ we perform the following queries.

(i) For each vertex $w$ in $p_0$, we query each path in $\mathcal{P}$ for an edge to vertex $w$. This takes $O(n|\mathcal{P}| \log^3 n)$ time. This query is irrespective of whether $p_0$ belonged to $P_p$ or $P_t$.

(ii) If $p_0$ belongs to $P_p$, then we query for an edge from all $\tau \in \mathcal{T}$ to $p_0$. Now from path halving we have that $|P_p|$ is bounded by $|\mathcal{P}| \log n$. Thus this step takes $O(n|\mathcal{P}| \log^4 n)$ time.

(iii) If $p_0$ belongs to $P_t$, then we query for an edge from only those sub-trees which were hanging from $p_0$. Note that these sub-trees 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 $O(n)$. Thus time total time taken for this step is $O(n \log^3 n)$.

Thus the total time taken by our algorithm is $O(n(1 + |\mathcal{P}| \log n) \log^3 n)$. So 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 $k \leq n$), the DFS tree of $G + U$ can be reported in $O(n(1 + |\mathcal{P}| \log n) \log^3 n)$ time.

From Definition 4.2, we have that $|\mathcal{P}|$ is bounded by $k$. 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 $k \leq n$), 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.

### 5.1 Extending algorithm to handle insertions

In order to update the DFS tree, our focus has been to restrict the number of edges to be processed. For the case when the updates are deletions only, we have been able to restrict this number to $O(nk \log n)$ if there are $k$ updates (failure of vertices or edges). If there are $k$ vertex insertions, the total number of edges introduced is anyway bounded by $nk$. So even if we incorporate all the insertions into the reduced adjacency
lists, the number of edges to be explored is going to remain bounded by $O(nk \log n)$. So the algorithm described above (only for deletions) can be extended to handle insertions as well by just initializing $L(v)$ as $\{ y \mid (y, v) \text{is a newly added edge}\}$ instead of $\emptyset$. This completes the proof of our main result - Theorem 1.1.

Let us consider the case when $U$ consists of edge insertions only. In this case $P$ will be an empty set. As discussed above, we will have to just initialize the reduced adjacency lists for the edge insertions in $U$. 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 $O(k + n \log^3 n)$ time.

### 6 Fully dynamic DFS

**Lemma 6.1** Let $P$ be any dynamic graph problem and $G$ be the input graph. Let $f, g, h$ be three functions of $m, n$ such that there exists a data structure $D$ for $P$ that (i) can be preprocessed in $f$ time, and (ii) given any set $U$ of $t$ ($t < n$) updates, is able to compute the solution for $G + U$ in $t \times g + h$ time. Then, there exists a dynamic algorithm for $P$ with worst case $O(\sqrt{fg} + h)$ update time.

**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 data structure $D_0$ over it. Now let $u_1, \ldots, u_c$ 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 $G_0 + \{u_1, \ldots, u_i\}$. This takes $h + (i \times g)$ time. So the total time for preprocessing and handling the first $c$ of updates is $f + \sum_{i=1}^{c} (h + (i \times g))$. 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 $c_0 = \sqrt{f/g}$. So, after every $c_0$ updates we rebuild our data structure and use it for next $c_0$ updates. (See Figure 5(a)). Substituting the value of $c_0$ gives the amortized time complexity as $O(\sqrt{fg} + h)$.

The above algorithm can be de-amortized as follows. Let $G_1, G_2, G_3, \ldots$ be the sequence of graphs obtained after $c_0, 2c_0, 3c_0, \ldots$ 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 $O(\sqrt{fg} + h)$.

![Figure 5](image.png)

Figure 5: (a) Obtaining an amortized algorithm from the data structure $D$ for handling $k$-updates. (b) Deamortization of the algorithm.

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

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 Applications

Our fully dynamic algorithm for maintaining a DFS tree can be used to solve various dynamic graph problems such as dynamic subgraph connectivity, bi-connectivity and 2 edge connectivity. Note that these problems are solved trivially using a DFS tree in the static setting. The solution of dynamic subgraph connectivity follows seamlessly from our fully dynamic algorithm as follows. As mentioned in Section 2, we maintain a DFS tree rooted at a dummy vertex $r$, such that the subtrees hanging from its children corresponds to connected components of the graph. Hence the connectivity query for any two vertices can be answered by comparing their ancestors at depth two (i.e. children of $r$). This information can be stored for each vertex and updated whenever the DFS tree is updated. Thus we have a data structure for subgraph connectivity with worst case $\tilde{O}(\sqrt{mn})$ update time and $O(1)$ query time. Similarly our algorithm can be used to solve dynamic subgraph bi-connectivity and 2-edge connectivity in the same time bounds. For details refer to Appendix A.5.

8 Conclusion

We have presented a fully dynamic algorithm for maintaining a DFS tree that takes worst case $\tilde{O}(\sqrt{mn})$ update time. This is the first fully dynamic algorithm that achieves $o(m)$ update time. In the fault tolerant setting our algorithm takes $\tilde{O}(nk)$ time to report a DFS tree, where $k$ is the number of vertex or edge failures in the graph. We show the immediate applications of fully dynamic DFS for solving various problems such as dynamic subgraph connectivity, bi-connectivity and 2 edge connectivity. We also prove the conditional lower bound of $\Omega(n)$ on maintaining DFS tree under vertex/edge updates. Refer to Appendix A.6 for details.

DFS tree has been extensively used for solving various graph problems in the static setting. Most of these problems are also solved efficiently in the dynamic environment. However, their solutions have not used dynamic DFS tree. This paper is an attempt to restore the glory of DFS trees for solving graph problems in the dynamic setting as was the case in the static setting. We believe that our dynamic algorithm for DFS, on its own or after further improvements/modifications, would encourage other researchers to use it in solving various other dynamic graph problems.

References

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014, 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 Shahbaz Khan. Incremental algorithm for maintaining DFS tree for undirected graphs. In Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Proceedings, Part I, pages 138–149, 2014.
[4] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. ACM Transactions on Algorithms, 8(4):35, 2012.

[5] J. L. Bentley. Solutions to Klee’s rectangle problems. Unpublished manuscript, Dept of Comp Sci, Carnegie-Mellon University, Pittsburgh PA, 1977.

[6] Timothy M. Chan. Dynamic subgraph connectivity with geometric applications. SIAM J. Comput., 36(3):681–694, 2006.

[7] Timothy M. Chan, Mihai Patrascu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. In 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA, pages 95–104, 2008.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms (3. ed.). MIT Press, 2009.

[9] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. J. ACM, 51(6):968–992, 2004.

[10] Ran Duan. New data structures for subgraph connectivity. In Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I, pages 201–212, 2010.

[11] Ran Duan and Seth Pettie. Connectivity oracles for failure prone graphs. In Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010, pages 465–474, 2010.

[12] 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.

[13] Shimon Even and Robert Endre Tarjan. Network flow and testing graph connectivity. SIAM J. Comput., 4(4):507–518, 1975.

[14] 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.

[15] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. SIAM J. Comput., 14(4):781–798, 1985.

[16] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. J. ACM, 46(4):502–516, 1999.

[17] 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.

[18] John E. Hopcroft and Richard M. Karp. An n5/2 algorithm for maximum matchings in bipartite graphs. SIAM J. Comput., 2(4):225–231, 1973.

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

[20] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In SODA, pages 1131–1141, 2013.
[21] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994.

[22] Mihai Patrascu and Mikkel Thorup. Planning for fast connectivity updates. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 263–271, 2007.

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

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

[25] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.

[26] Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012.

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

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

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

[30] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Inf.*, 6:171–185, 1976.

[31] Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.
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: It is sufficient to show that whenever a vertex $v$ is popped out, i.e. it has finished scanning $L(v)$, all the neighbors of $v$ must have been visited by the traversal. Suppose this does not hold true for the procedure Dynamic-DFS. Let $w$ be the first vertex popped out from the stack for which this condition is violated, and let $b$ be its neighbor which remains unvisited when $w$ is popped.

As follows from the description of the procedure Dynamic-DFS, computation of the updated DFS tree is carried out by attaching paths to a partially constructed DFS tree. Let $p$ be the path attached to the tree to which vertex $w$ belonged. Consider the graph left unvisited by the traversal just after the path $p$ is attached. Let $C_0$ be the connected component containing vertex $b$ in this unvisited graph.

Let $u$ be the lowest vertex on $p$ which has an edge incident from $C_0$. If the procedure Dynamic-DFS maintains Property 5.1 at least one vertex (say $a$) from $N(u) \cap C_0$ must be added to $L(u)$. Now since $u$ must be either equal to $w$ or a descendant of $u$, we have that $a$ must have been visited by the traversal before $w$ is popped out. Now let $A$ be the set of the vertices in $C_0$ which are visited by the traversal before $w$ is popped out, and $B$ be the subset of vertices which are left unvisited. Then $A$ and $B$ are both non empty since $a \in A$ and $b \in B$. Note that all the vertices in set $A$ must be popped out before $w$ is popped out. Thus by the definition of $w$, we have that for each $y \in A$, $N(y)$ must have been visited by the traversal before $y$ is popped out. Since $B$ is not visited by the traversal, $A$ and $B$ are not connected. This contradicts our assumption that $C_0$ is a connected component.

□
A.2 Pseudo code for DFS-in-Tree and DFS-in-Path

**Procedure** DFS-in-Tree($u$): DFS traversal enters from node $u$ and exits from $v$, the root of the tree containing node $u$ in set $\mathcal{T}$.

1. $v \leftarrow u$;
2. **while** ISROOT($v$) $\neq$ True **do**
   3. $v \leftarrow \text{par}(v)$
   4. **end**
   5. ISROOT($v$) $\leftarrow$ False;
   6. $\mathcal{T} \leftarrow \mathcal{T} \cup \{v\}$
   7. $(w_1, \ldots, w_t) \leftarrow \text{path}(u, v)$;
   8. **for** $i = 1$ **to** $t$ **do**
      9. $\text{Push}(w_i)$;
      10. status($w_i$) $\leftarrow$ visited;
      11. **if** $i \neq 1$ **then** $\text{par}(w_i) \leftarrow w_{i-1}$
      **foreach** path($x, y) \in P$ **do**
         12. **if** $\text{Query}(w_i, x, y) \neq \emptyset$ **then**
            13. $(w_i, z) \leftarrow \text{Query}(w_i, x, y)$;
            14. $L(w_i) \leftarrow L(w_i) \cup \{z\}$;
      **end**
   **end**
   **foreach** child $w$ of $w_i$ except $w_{i-1}$ **do**
   17. $(y, z) \leftarrow \text{Query}(T(w), v, u)$;
      /* where $y \in \text{path}(u, v)$ */
   18. $L(y) \leftarrow L(y) \cup \{z\}$
   19. $\mathcal{T} \leftarrow \mathcal{T} \cup T(w)$
   20. ISROOT($w$) $\leftarrow$ True;
   **end**

**Procedure** DFS-in-Path($u, v, d$): DFS traversal enters path($v, d$) from node $u$ and exits from $v$ or $d$ whichever is farther from $u$.

1. **if** $\text{dist}_{\mathcal{T}}(u, d) > \text{dist}_{\mathcal{T}}(u, v)$ **then** $\text{Swap}(v, d)$
   2. $c \leftarrow$ Neighbor of $u$ on path($v, d$) nearer to $d$;
   3. $P \leftarrow (P \setminus \text{path}(v, d)) \cup \text{path}(c, d)$;
   4. **for** $c' \in \text{path}(c, d)$ **do**
      5. $\text{PPathParam}(c') \leftarrow (c, d)$;
   **end**
   6. $(w_1, \ldots, w_t) \leftarrow \text{path}(u, v)$;
   7. **for** $i = 1$ **to** $t$ **do**
      8. $\text{Push}(w_i)$;
      9. status($w_i$) $\leftarrow$ visited;
      10. **if** $i \neq 1$ **then** $\text{par}(w_i) \leftarrow w_{i-1}$**foreach** path($x, y) \in P$ **do**
            11. **if** $\text{Query}(w_i, x, y) \neq \emptyset$ **then**
                12. $(w_i, z) \leftarrow \text{Query}(w_i, x, y)$;
                13. $L(w_i) \leftarrow L(w_i) \cup \{z\}$;
            **end**
      **end**
   17. **foreach** $T(w) \in \mathcal{T}$ **do**
      18. **if** $\text{Query}(T(w), v, u) \neq \emptyset$ **then**
            19. $(y, z) \leftarrow \text{Query}(T(w), v, u)$;
               /* where $y \in \text{path}(u, v)$ */
            20. $L(y) \leftarrow L(y) \cup \{z\}$;
      **end**
   **end**

A.3 Proof of Correctness

To prove the correctness of our algorithm we ensure that each time the DFS traversal returns from the DFS-in-Tree or DFS-in-Path procedure than the following two invariants holds true.

1. The vertices in trees and paths belonging to the set $\mathcal{T} \cup P$ are precisely the vertices left unvisited by the DFS traversal.

2. Let $p_0$ be the path attached to the partially constructed DFS tree in the last call of DFS-in-Tree or DFS-in-Path procedure. Then for each connected component $C$ in the unvisited graph we have that if $w$ is the lowest vertex on $p_0$ adjacent to $C$ then at least one vertex from $N(w) \cap C$ is added to the set $L(w)$. (Property 5.1)

Note that from the sufficiency of Property 5.1 it follows that if the above invariants are maintained then our algorithm will output a correct DFS tree.
Proof for Invariant 1: We first consider the DFS-in-Tree procedure. In this procedure, the DFS traversal enters a tree \( T(v) \) from node \( u \) and follows the path till root \( v \). This tree is removed from \( T \), and the subtrees hanging from this path are added back to the set \( T \). Similarly if DFS-in-Path procedure is called, then the original path is removed from \( P \) and the unvisited segment of it is added back to set \( P \). Therefore, it is easy to see that Invariant 1 always holds true.

Proof for Invariant 2: We first prove the Invariant 2 for the case when DFS traversal enters a path in \( P \). Let \( p_0 \) be the segment of this path traversed by the DFS traversal. From Invariant 1 we have that once \( p_0 \) is attached to tree \( T^* \), the paths and trees belonging to \( T \cup P \) represent the unvisited graph. Consider an element \( C \) in \( T \cup P \). We have the following two cases -

1. If \( C \) is a tree and \( w \) is the lowest vertex on \( p_0 \) adjacent to \( C \), then at least one vertex from \( N(w) \cap C \) is added to the set \( L(w) \). (See steps 19-24 of DFS-in-Path procedure).

2. If \( C \) is a path then for each \( w \) (and hence the lowest one also) on \( p_0 \) adjacent to \( C \), we add one vertex from \( N(w) \cap C \) to the set \( L(w) \). (See steps 12-17 of DFS-in-Path procedure).

Now consider the case when DFS tree enters a tree \( T(v) \) in \( T \). The proof that Invariant 2 is satisfied in this case can be done in a similar manner except that instead of processing all the trees only the subtrees hanging from \( p_0 \) are processed. Here we exploit the fact that the trees which were already present in \( T \) cannot have an edge incident to \( p_0 \). This is due to the fact that there are no cross edges in a DFS tree, and the trees in \( T \) are subtrees of the original DFS tree \( T \).

A.4 Data Structure

The efficiency of our dynamic algorithm relies heavily on the the data structure \( D \). We now describe it in details. Its construction employs a novel combination of two well known techniques, namely, heavy-light decomposition and suitably augmented segment tree.

We perform a special DFS traversal on the tree \( T \) in which we traverse children of a vertex in the decreasing order of the size of the subtrees hanging from them. Let \( L \) be the list of the vertices of \( T \) arranged in the order of their visit time during this traversal. For notational convenience henceforth, a vertex will also be denoted by its visit-number. Thus \( L[i] \) is equal to vertex \( i \). Also \( L[i,j] \) denotes the sequence of vertices from \( i \) to \( j \). The special DFS traversal ensures that any path in \( T \) can be represented by \( O(\log n) \) contiguous sequences of vertices (i.e. vertices having contiguous visit times). This property was earlier used in heavy-light decomposition by Sleator and Tarjan [27] for dynamic trees.

In order to answer the queries on \( T \), we define two additional queries on list \( L \). Let \( i,j,p,q \) be integers in the range 1 to \( n \) satisfying the relation \( i \leq j < p \leq q \). The two queries on \( L \) are -

- \( Q_F(i,j,p,q) \): Reports an edge \((a,b)\) s.t. \( a \) is the first vertex in list \( L[i,j] \) having an edge incident on it from some \( b \) in \( L[p,q] \).
- \( Q_L(i,j,p,q) \): Reports an edge \((a,b)\) s.t. \( a \) is the last vertex in list \( L[i,j] \) having an edge incident on it from some \( b \) in \( L[p,q] \).

**Lemma A.2** The two queries, \( Query(T(w),x,y) \) and \( Query(w,x,y) \), on the DFS tree \( T \) can be answered using at most \( O(\log n) \) queries \( Q_F \) and \( Q_L \) on list \( L \).

**Proof:** Consider the query \( Query(T(w),x,y) \) on tree \( T \). Let \( p \) and \( q \) be the visit time of first and last visited vertices in the subtree \( T(w) \). Then \( L[p,q] \) is the contiguous sequence corresponding to the vertices in subtree \( T(w) \) (see Figure 6). For each vertex \( w \), we can precompute these two numbers \( p \) and \( q \) during preprocessing. Let \( path(x_1,y_1),...,path(x_k,y_k) \) be \( k \) contiguous paths of \( T \) that constitute
path(x, y). Note that \( k = O(\log n) \) as follows from the heavy-light decomposition described above. The query \( Q(T(w), x, y) \) can thus be answered by finding the edge closest to \( x \) among the edges reported by \( Q(T(w), x_1, y_1), \ldots, Q(T(w), x_k, y_k) \). Further let \( \mathcal{L}[p_i, q_i] \) represent the sequence of vertices of \( \text{path}(x_i, y_i) \) for each \( 1 \leq i \leq k \). It can be observed that each query \( Q(T(w), x_i, y_i) \) is equivalent to the query \( Q_H(p, q, p_i, q_i) \) in case \( x \) in an ancestor of \( y \) and \( Q_L(p, q, p_i, q_i) \) otherwise. Similarly \( Q(w, x_i, y_i) \) is equivalent to the query \( Q_H(p, p_i, q_i) \) in case \( x \) in an ancestor of \( y \) and \( Q_L(p, p_i, q_i) \) otherwise. \( \square \)

![DFS tree](image.png)

Figure 6: (i) The highest edge from subtree \( T(w) \) on \( \text{path}(x, y) \) is edge \((1, 4)\) and the lowest edges are edge \((2, 3)\) and \((2, 5)\). (ii) The vertices of \( T(w) \) are represented as union of two subtrees in segment tree \( \mathcal{T}_B \).

We now describe an efficient data structure to answer the queries \( Q_F \) and \( Q_L \) on list \( \mathcal{L} \). Let \( \mathcal{T}_B \) be a segment tree\[^5\] whose leaf nodes from left to right represent the vertices in list \( \mathcal{L} \) (see Figure 6). We know that any interval \([p, q]\) of leaves of \( \mathcal{T}_B \) can be represented by union of \( O(\log n) \) disjoint subtrees in \( \mathcal{T}_B \). Consider an edge \((u, v)\) in \( G \). Both \( u \) and \( v \) are leaf nodes in \( \mathcal{T}_B \) and have \( O(\log n) \) ancestors each. We store the edge as the ordered pair \((u, v)\) at all ancestors of \( v \) (including itself) in tree \( \mathcal{T}_B \). Similarly we store the edge as the ordered pair \((v, u)\) at all the ancestors of \( u \) (including itself) in tree \( \mathcal{T}_B \). Thus each edge in \( G \) is stored at \( O(\log n) \) levels in \( \mathcal{T}_B \). Let \( E(b) \) be the collection of edges stored at any node \( b \) in \( \mathcal{T}_B \). We keep the sets \( E \) of edges sorted in the lexicographic order as a balanced binary search tree.

We now describe how query \( Q_H(i, j, p, q) \) is answered by \( \mathcal{T}_B \) (\( Q_L(i, j, p, q) \) can be answered in a similar manner):

Let \( T_1, \ldots, T_\ell \) be the subtrees in \( \mathcal{T}_B \) representing the range \( \mathcal{L}[i, j] \). For each \( \alpha, 1 \leq \alpha \leq \ell \), we find the predecessor of \((q, n + 1)\) in the set \( E(a) \) where \( a \) is the root of \( T_\alpha \). This requires a predecessor query on the binary search tree stored at \( a \). We return the lexicographically largest ordered pair among these edges reported by these \( \ell \) predecessor queries.

Since in a binary search tree any predecessor or successor query can be answered in \( O(\log n) \) time, a query \( Q_L \) or \( Q_H \) can be answered in \( O(\log^2 n) \). The space complexity is \( O(m \log n) \) as each edge is stored at \( O(\log n) \) levels. Hence the total time required to create \( \mathcal{T}_B \) and a BST on \( E(b) \) for each node \( b \) in \( \mathcal{T}_B \) is \( O(m \log n) \). Hence we have the following lemma.

**Lemma A.3** The queries \( Q_H, Q_L \) on list \( \mathcal{L} \) can be answered in \( O(\log^2 n) \) worst case time by using \( O(m \log n) \) space data structures. The preprocessing time, i.e. the time for construction of the data structure is \( O(m \log n) \).

Using Lemma A.2 and Lemma A.3 we have the following main theorem.

**Theorem A.1** The queries \( \text{Query}(T(w), x, y) \), \( \text{Query}(w, x, y) \) on \( T \) can be answered in \( O(\log^3 n) \) worst case time by using \( O(m \log n) \) space data structure \( \mathcal{D} \). The time for the construction of the data structure is \( O(m \log n) \).
Lemma A.4 Let \( E_D \) be a set of edges in \( G \) of size \( O(k) \). Then the edges \( E_D \) can be deleted from the data structure \( D \) in time \( O(k \log^2 n) \) time.

Proof: Each edges \( e \in E_D \) is stored in \( O(\log n) \) binary search trees corresponding to \( E \) of \( O(\log n) \) nodes of \( T_0 \). Deleting an edge from a binary search tree takes \( O(\log n) \) time. Note that these binary search trees may end up skewed after these deletions, however the height of each tree will still be bounded by \( O(\log n) \). Hence, the total time required is \( O(\log^2 n) \) time per edge deletion.

A.5 Applications

In this section we show how fully dynamic DFS can be used to solve dynamic subgraph bi-connectivity and 2-edge connectivity.

A set \( S \) is said to be a biconnected component if it is maximal set of vertices such that on failure of any vertex \( w \) in \( S \), the vertices of \( S \setminus \{w\} \) remains connected. Similarly, a set \( S \) is said to be 2-edge connected if it is maximal set of vertices such that failure of any edge with both end points in \( S \) does not disconnect any two vertices in \( S \). The goal is to maintain a data structure that can efficiently answer the queries of bi-connectivity and 2-edge connectivity in the dynamic setting.

These queries can be answered easily by finding articulation points and bridges of the graph. It can be shown\(^8\) that two vertices belong to same biconnected component if and only if the DFS tree path between them does not passes through an articulation point. Similarly, two vertices belong to same 2-edge connected component if and only if the DFS tree path between them does not have a bridge. The articulation point and bridge in a graph can be defined as follows:

Definition A.1 Given a graph \( G(V,E) \), a vertex \( v \in V \) is called an articulation point of \( G \) if there exist a pair of vertices \( x,y \in V \) such that every path between \( x \) and \( y \) in \( G \) passes through \( v \).

Definition A.2 Given a graph \( G(V,E) \), an edge \( e \in E \) is called a bridge of \( G \) if there exist a pair of vertices \( x,y \in V \) such that every path between \( x \) and \( y \) in \( G \) passes through \( e \).

The articulation points and bridges of a graph can be easily computed by using DFS traversal of the graph. Given a DFS tree \( T \) of an undirected graph \( G \), we can index the vertices in order they were visited by the DFS traversal. This index is called the DFN number of the vertex. The high number of a vertex \( v \) is defined as the lowest DFN number vertex from which there is an edge incident to \( T(v) \). Now any non-root vertex \( v \) will be an articulation point of the graph if high number of at least one of its children is equal to \( DFN(v) \). The root \( r \) of the DFS tree \( T \) will be an articulation point if it has more than one children. Any edge \( (x,y) \) (say \( x = \text{par}(y) \)) of the DFS tree will be a bridge if either \( y \) is a leaf node, or the high number of each child of \( y \) is equal to \( y \). Thus given the high number of each vertex in the DFS tree, the articulation points and bridges can be determined in \( O(n) \) time.

We now show that if there are \( k \) updates to \( G \), then we can not only construct the new tree \( T^* \) in \( O(nk \log^4 n) \) time, but also compute high point of each vertex in the same time bound. For each vertex \( x \), let \( a(x) \) denote the highest ancestor of \( x \) in \( T^* \) such that \( (x,a(x)) \) is an edge in \( G+U \). Note that if \( (x,a(x)) \) is a newly added edge, then it can be easily computed by scanning all the new edges added to the graph. This is due to fact that the total number of new edges added to \( G \) is bounded by \( nk \). So we restrict ourselves to the case when \( (x,a(x)) \) was originally present in the graph \( G \). Recall that our algorithm computed \( T^* \) by attaching paths to the partially grown tree. Let \( P_l \) be the set of paths attached to \( T^* \) (during its construction) that originally belonged to some tree in set \( T \). Similarly, let \( P_p \) be the set of paths attached to \( T^* \) that originally belonged to some path in set \( P \). Note that path halving ensures that the size of \( P_p \) is bounded by \( k \log n \). For each path \( p_0 \in P_l \cup P_p \), let \( H(p_0) \) denote the vertex \( p_0 \) that is closest to \( r \) in \( T^* \).

We now give an \( O(nk \log^4 n) \) time algorithm for constructing a subset \( A(x) \) of neighbors \( x \) such that the following condition holds.
For a vertex \( x \), if \( a(x) \notin A(x) \), then there is some descendant \( y \) of \( x \) in \( T^* \) such that \( a(x) \in A(y) \).

It is easy to see that if we get such an \( A(x) \) corresponding to each \( x \), then high points can be computed easily by processing the vertices of \( T^* \) in bottom-up manner. Now depending upon whether paths containing \( x \) and \( a(x) \) belong to set \( P_p \) or \( P_t \), we can have different cases. We show the construction of set \( A(x) \) along with the case it handles in following four steps.

1. Vertex \( a(x) \) lies on a path in \( P_p \)

Consider a vertex \( x \) in \( T^* \). For each path \( p_0 \in P_p \) we query our data structure for an edge \((u, x)\) such that the end point \( u \) is closest to \( H(p_0) \) on path \( p_0 \), and add \( u \) to \( A(x) \). Note that if \( a(x) \) lies on \( p_0 \), then \( u \) will be same as \( a(x) \).

2. Vertex \( x \) lies on a path in \( P_p \)

For each \( u \in T^* \) and \( p_0 \in P_p \), we query our data structure for an edge \((u, y)\) such that the end point \( y \) is farthest from \( H(p_0) \) on path \( p_0 \). We add \( u \) to \( A(y) \). Now consider a vertex \( x \) on \( p_0 \) such that \( a(x) = u \). If \( x \) is equal to \( y \), then we have added \( a(x) \) (i.e. \( u \)) to \( A(x) \). If \( x \) is not equal to \( y \), then we have added \( a(x) \) (i.e. \( u \)) to \( A(y) \) where \( y \) is descendant of \( x \) in \( T^* \).

3. Vertex \( x \) and \( a(x) \) lies on same path in \( P_t \)

Consider a vertex \( x \) lying on path \( p_0 \in P_t \). We query our data structure for an edge \((u, x)\) such that the end point \( u \) is closest to \( H(p_0) \) on path \( p_0 \), and add \( u \) to \( A(x) \). Note that if \( a(x) \) also lies on \( p_0 \), then \( u \) will be same as \( a(x) \).

4. Vertex \( x \) and \( a(x) \) lies on different paths in \( P_t \)

We first note that \( x \) and \( a(x) \) would belong to same tree only (say \( T_0 \)) since by our tree partitioning we have that an edge originally present in \( G \) cannot have its end point in different trees in \( T \). Let \( z \) be the highest ancestor of \( x \) in \( T_0 \) such that \((x, z)\) is an edge in \( G + U \). Let \( p_z \) be the path in \( P_t \) containing vertex \( z \).

We first show that \( a(x) \) must belong to \( p_z \). Recall that as the algorithm proceeds, our partitioning \( P \cup T \) evolves with time. Let \( T_1 \) be the tree in \( T \) containing vertex \( z \) just before \( p_z \) is attached to \( T^* \). Then \( T_1 \) must be either equal to \( T_0 \), or a subtree of \( T_0 \). (See Figure 7). Also \( a(x) \) must lie in tree \( T_1 \), since it cannot be an ancestor of \( z \) in \( T_0 \). Now let \( T_2 \) be the tree containing \( x \) which is obtained on removal of \( p_z \) from \( T_1 \). Note that vertices in \( T_2 \) must be hanging either from vertex \( z \) or some descendant of \( z \) in \( T^* \). Hence \( a(x) \) must belong to \( p_z \).

Therefore for each vertex \( x \) belonging to a tree \( T_0 \) in \( T \), we calculate the highest ancestor of \( z \) of \( x \) in \( T_0 \) which has an edge incident from \( T_0 \). Now once \( T^* \) is constructed let \( p_z (\in P_t) \) be the path containing vertex \( z \). We query our data structure for an edge \((u, x)\) such that the end point \( u \) is closest to \( H(p_z) \) on path \( p_z \), and add \( u \) to \( A(x) \). Note that if \( a(x) \) also lies in \( T_0 \), then \( u \) must be same as \( a(x) \). (See Figure 7).

Note that the total time taken by the first two steps is bounded by \( O(nk \log^4 n) \) and by the last two steps is bounded by \( O(n \log^2 n) \). Thus we have the following theorem.

**Theorem A.2** Given an undirected graph \( G(V, E) \) with \( |V| = n \) and \( |E| = m \), we can maintain a data structure for answering queries of biconnected components and 2 edge connectivity in a dynamic graph which takes \( O(\sqrt{mn \log^{2.5} n}) \) update time, \( O(1) \) query time and \( O(m \log n) \) time for preprocessing.

## A.6 Lower Bounds

We now prove two conditional lower bound for maintaining DFS tree under vertex updates and edge updates.
Figure 7: (i) Before the beginning of algorithm vertex $x$ belongs to tree $T_0 \in \mathcal{T}$, $z$ is highest ancestor of $x$ in $T_0$ such that $(x, z)$ is an edge. (ii) The partitioning changes as the algorithm proceeds, $T_1(\in \mathcal{T})$ is the tree containing vertex $z$ just before it is attached to $T^*$. (iii) A path containing vertex $z$ (i.e. $p_z$) is extracted from $T_1$ and attached to $T^*$. If $a(x)$ belonged to $T_0$, then it is equal to vertex $u$.

**Vertex Updates**

The bounds for maintaining DFS tree under vertex updates are based on Strong Exponential Time Hypothesis (SETH) as defined below:

**Definition A.3 (SETH)** For every $\epsilon > 0$, there exists a $k$, such that SAT on $k-CNF$ formulas on $n$ variables cannot be solved in $\tilde{O}(2^{(1-\epsilon)n})$ time.

Given an undirected graph $G$ on $n$ vertices and $m$ edges in a dynamic environment (incremental / decremental or fully dynamic) under vertex updates. The status of any vertex can be switched between active and inactive in an update. The goal of subgraph connectedness is to efficiently answer whether the subgraph induced by active vertices is connected. Abboud and Williams [1] proved a conditional lower bound for maintaining subgraph connectedness of $\Omega(n)$ based on SETH. They proved that any algorithm for maintaining subgraph connectedness using arbitrary polynomial preprocessing time and $O(n^{1-\epsilon})$ update time would essentially refute the SETH conjecture.

We present a reduction from subgraph connectedness to maintaining DFS tree under vertex updates requiring the algorithm to report whether the number of children of the root in any DFS tree of the subgraph is greater than 1. Thus we establish the following:

**Theorem A.3** Given an undirected graph $G$ with $n$ vertices and $m$ edges undergoing vertex updates, an algorithm for maintaining DFS tree (that can report the number of children of the root in the DFS tree) with preprocessing time $p(m, n)$, update time $u(m, n)$ and query time $q(m, n)$ would imply an algorithm for subgraph connectedness with preprocessing time $p(m+n, n)$, update time $u(m+n, n)$ and query time $q(m+n, n)$.

**Proof:** Given the graph $G$ for which we need to query for subgraph connectedness, we make a graph $G'$ as follows. We add all vertices and edges of $G$ to $G'$. Further add another vertex $r$ called as pseudo root and connect it to all other vertices of $G'$. Thus $G'$ has $n+1$ vertices and $m+n$ edges. Now in any DFS tree $T'$ of $G'$. rooted on $r$, the number of children of $r$ will be equal to the number of components in $G$. Here subtrees rooted on each child of $s$ represents a component of $G$. Any change on $G$ can be performed
on $G'$ and query for subgraph connectedness in $G$ is equivalent to querying if $r$ has more than 1 child in $T$. □

Thus any algorithm for maintaining DFS under vertex updates with arbitrary preprocessing time and $O(n^{1-\epsilon})$ update time would refute SETH.

**Edge Updates**

The bounds for maintaining DFS tree under edge updates applies on any algorithm that maintains tree edges of the DFS tree explicitly. In the following example we prove that there exists a graph $G$ and a sequence of edge updates $U$ such that any DFS tree of the graph have $O(n)$ of its edges converted from tree edges to back edges and vice-versa.

![Figure 8: Worst Case Example for lower bound](image)

Consider the following graph for which DFS tree rooted at $v_1$ is to be maintained under fully dynamic edge updates. There are $n/2$ vertices $u_1, \ldots, u_{n/2}$ that have edges to vertices $x$ and $y$. The remaining $n/2 - 2$ vertices $v_1, \ldots, v_k$ are connected in form of a line as shown in Figure 8. At any point of time one of $v_1, \ldots, v_k$ is connected to either $x$ or $y$. Figure 8 shows $v_1$ connected to $y$. The DFS tree for the graph is shown in Figure 8(a). Now insertion of edge $(v_i, x)$ (say $i = 2$) and deletion of edge $v_1, y$, will transform the DFS tree to either Figure 8(c) or Figure 8(c). Clearly $O(n)$ edges are changed from tree edge to back edge and vice-versa. This can happen for any $v_i$ alternating between $x$ and $y$. Thus any algorithm maintaining tree edges explicitly undergoes would take $\Omega(n)$ time to handle such a pair of edge updates.