On Dynamic DFS Tree in Directed Graphs *

Surender Baswana ** and Keerti Choudhary ***

Department of CSE, IIT Kanpur,
Kanpur-208016, India
{sbaswana,keerti}@cse.iitk.ac.in
http://www.cse.iitk.ac.in

Keywords: dynamic, decremental, directed, graph, depth first search

Abstract. Let $G = (V, E)$ be a directed graph on n vertices and m edges. We address the problem of maintaining a depth first search (DFS) tree efficiently under insertion/deletion of edges in G.

1. We present an efficient randomized decremental algorithm for maintaining a DFS tree for a directed acyclic graph. For processing any arbitrary online sequence of edge deletions, this algorithm takes expected $O(mn \log n)$ time.
2. We present the following lower bound results.
   (a) Any decremental (or incremental) algorithm for maintaining the ordered DFS tree *explicitly* requires $\Omega(mn)$ total update time in the worst case.
   (b) Any decremental (or incremental) algorithm for maintaining the ordered DFS tree is at least as hard as computing all-pairs reachability in a directed graph.

1 Introduction

Depth First Search (DFS) is a well known graph traversal technique. Tarjan, in his seminal work [14], demonstrated the power of DFS traversal for solving various fundamental graph problems, namely, connected components, topological sorting and strongly connected components.

A DFS traversal is a recursive algorithm to traverse a graph. Let $G = (V, E)$ be a directed graph on $n = |V|$ vertices and $m = |E|$ edges. The DFS traversal carried out in G from a vertex $r \in V$ produces a tree, called DFS tree rooted at r. This tree spans all the vertices reachable from r in G. It takes $O(m + n)$ time to perform a DFS traversal and generate its DFS tree. A DFS tree leads

* Full version of this article is available at http://www.cse.iitk.ac.in/users/sbaswana/Papers-published/Dynamic-DFS-digraph.pdf.

** This research was partially supported by the India-Israel joint research project on dynamic graph algorithms, and 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.

to a classification of all non-tree edges into three categories, namely, *back* edges, *forward* edges, and *cross* edges. Most of the applications of a DFS tree exploit the relationship among the edges based on this classification. For a given graph, there may exist many DFS trees rooted at a vertex r. However, if the DFS traversal is performed strictly according to the adjacency lists, then there will be a unique DFS tree rooted at r. The ordered DFS problem is to compute the order in which the vertices get visited during this restricted traversal.

Most of the graph applications in real life deal with a graph that is not static. Instead, the graph keeps changing with time - some edges get deleted while some new edges get inserted. The dynamic nature of these graphs has motivated researchers to design efficient algorithms for various graph problems in a dynamic environment. Any algorithmic graph problem can be modeled in the dynamic environment as follows. There is an online sequence of insertion and/or deletion of edges, and the aim is to update the solution of the graph problem efficiently after each edge insertion/deletion. There exist efficient dynamic algorithms for various fundamental problems in graphs [4, 7, 12, 13, 15]. In this article we address the problem of maintaining a DFS tree in a dynamic graph.

Though an efficient algorithm is known for the static version of the DFS tree problem, the same is not true for its dynamic counterpart. Reif [9, 10] and Miltersen et al. [8] addressed the complexity of the ordered DFS problem in a dynamic environment. Miltersen et al. [8] introduced a class of problems called non-redundant polynomial (NRP) complete. They showed that if the solution of any NRP-complete problem is updatable in $O(\text{polylog}(n))$ time, then the solution of every problem in the class P is updatable in $O(\text{polylog}(n))$ time. The ordered DFS tree problem was shown to be NRP-complete. So it is highly unlikely that any $O(\text{polylog}(n))$ update time algorithm would exist for the ordered DFS problem in the dynamic setting.

Apart from showing the hardness of the ordered DFS tree problem, very little work has been done on the design of any non-trivial algorithm for the problem of maintaining any DFS tree in a dynamic environment. Franciosa et al. [6] designed an algorithm for maintaining a 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. This incremental algorithm is the only non-trivial result known for the dynamic DFS tree problem in directed graphs. For the related problem of maintaining a breadth first search (BFS) tree in directed graphs, Franciosa et al. [5] designed a decremental algorithm that achieves $O(n)$ amortized time per edge deletion.

In this article, we complement the existing upper and lower bound results for the dynamic DFS tree problem in a directed graph. Our main result is the first non-trivial decremental algorithm for a DFS tree in a DAG.

1.1 An efficient decremental algorithm for a DFS tree in a DAG

We present a decremental algorithm that maintains a DFS tree in a DAG and takes expected $O(mn \log n)$ time to process any arbitrary online sequence of

edge deletions. Hence the expected amortized update time per edge deletion is $O(n \log n)$. We now provide an overview of this randomized algorithm.

Consider the deletion of a tree edge, say (a, b), in a DFS tree. It may turn out that each vertex of subtree $T(b)$ may have to be *hung* from a different parent in the new DFS tree (see Figure 1). This is because the parent of a vertex in a DFS tree depends upon the order in which its in-neighbors get visited during the DFS traversal.

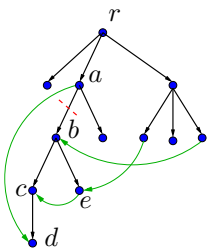

Fig. 1. Deletion of edge (a, b) results in the change of parent of every vertex in $T(b)$.

In order to achieve an efficient update time, we use randomization in the DFS traversal. Once the traversal reaches a vertex v, the next vertex to be traversed is selected randomly uniformly among all unvisited neighbors of v. Our decremental algorithm maintains this *randomized* DFS tree at each stage. This randomization plays a crucial role in fooling the adversary that generates the sequence of edge deletions: For a pair of vertices, say u and x, the deletion of very few (in expectation) outgoing edges of u will disconnect the tree path from root to x. Hence a vertex will have to search for a new parent *fewer* times during any given sequence of edge deletions. We analyze this algorithm using probability tools and some fundamental properties of a DFS tree in a DAG. This leads us to achieve an upper bound of $O(mn \log n)$ on the expected running time of this algorithm for any arbitrary sequence of edge deletions.

The DFS tree maintained (based on the random bits) by the algorithm at any stage is not known to the adversary for it to choose the updates adaptively. This oblivious adversarial model is no different from randomized data structures like universal hashing.

1.2 Lower bounds on dynamic DFS tree

We get the following lower bound results for maintaining an ordered DFS tree.

1. The most common way of storing any rooted tree is by keeping a parent pointer for each vertex in the tree. We call this representation an *explicit* representation of a tree. We establish a worst case lower bound of $\Omega(mn)$ for maintaining the ordered DFS tree explicitly under deletion (or insertion) of edges. This lower bound holds for undirected as well as directed graphs. Recently, an $O(n^2)$ time incremental algorithm [2] has been designed for maintaining a DFS tree explicitly in any undirected graph. Therefore, our lower bound result implies that for dense graphs (when $m = \Theta(n^2)$) maintaining a DFS tree explicitly in the incremental environment is provably faster than maintaining an ordered DFS tree by a factor of $\Omega(n)$.
2. We show that the dynamic DFS tree problem in a directed graph is closely related to the static all-pairs reachability problem. We provide an $O(m + n)$

time reduction from the static all-pairs reachability problem to the decremental (or incremental) maintenance of the ordered DFS tree in a graph G' with $O(m)$ edges and $O(n)$ vertices. This reduction is similar to the reduction technique used by Roditty and Zwick [11] for the decremental (or incremental) BFS tree problem. This reduction implies conditional lower bounds of $\Omega(\min(mn, n^{\omega}))$ on the total update time and $\Omega(\min(m, n^{\omega-1}))$ on the worst case update time per edge deletion for any decremental (or incremental) algorithm for the ordered DFS tree problem. Here ω is the exponent of the fastest matrix multiplication algorithm, and currently $\omega < 2.373$ [16].

1.3 Organization of the paper

We describe notations and terminologies related to a DFS tree in Section 2. In Section 3, we first describe a deterministic algorithm for maintaining a DFS tree in a DAG under deletion of edges. This algorithm is very simple. However, in the worst case it can take $\Theta(m^2)$ time. In Section 4, we show that by adding a small amount of randomization to this algorithm, its expected running time gets reduced to $O(mn \log n)$. In Section 5, we provide lower bounds on the dynamic DFS tree problem.

2 Preliminaries

Given a directed acyclic graph $G = (V, E)$ on $n = |V|$ vertices and $m = |E|$ edges, and a given root vertex $r \in V$, the following notations will be used throughout the paper.

- T : A DFS tree of G rooted at r.
- START-TIME(x): The time at which the traversal reaches x for the first time when we carry out the DFS traversal associated with T.
- FINISH-TIME(x): The time at which the DFS traversal finishes for vertex x.
- $\deg(x)$: The number of edges entering into vertex x.
- $IN(x)$: The list of vertices of T having an outgoing edge into x; this list is sorted in the topological ordering.
- $OUT(x)$: The list of vertices of T having an incoming edge from x.
- $par(x)$: The parent of x in T.
- $path(x)$: The path from r to x in T.
- LEVEL(x) : The level of a vertex x in T such that LEVEL(r) $= 0$ and LEVEL(x) $=$ LEVEL($par(x)$) $+ 1$.
- $T(x)$: The subtree of T rooted at a vertex x.
- $LCA(x, y)$: The Lowest Common Ancestor of x and y in tree T.
- $LA(x, k)$: The ancestor of x at level k in tree T.

Our algorithm uses the following results for the dynamic version of the Lowest Common Ancestor (LCA) and the Level Ancestors (LA) problems.

Theorem 1 (Cole and Hariharan 2005[3]). *There exists a data structure for maintaining a rooted tree using linear space that can answer any LCA query in $O(1)$ time and can handle insertion or deletion of a leaf node in $O(1)$ time.*

Theorem 2 (Alstrup and Holm 2000[1]). *There exists a data structure for maintaining a rooted tree using linear space that can answer any level ancestor query in $O(1)$ time and can handle insertion of a leaf node in $O(1)$ time.*

The data structure for the Level Ancestor problem can be easily extended to handle deletion of a leaf node in amortized $O(1)$ time using the standard technique of periodic rebuilding. The following lemma, with a straight forward proof, states two properties of the DFS traversal in a DAG. This lemma will play a crucial role in our randomized decremental algorithm for DFS tree.

Lemma 1. *Let y and z be any two vertices in a given directed acyclic graph G. If there is a path from y to z, then the following two properties hold true for each DFS traversal carried out in G.*

P1 : FINISH-TIME*(z)* $<$ FINISH-TIME*(y)*.
P2 : *If* START-TIME*(y)* $<$ START-TIME*(z), then z must be a descendant of y in the DFS tree.*

3 A simple and deterministic decremental algorithm

We shall now present a deterministic and very simple algorithm for maintaining the ordered DFS tree defined by a fixed permutation of vertices.

Let σ be a permutation of V (a bijective mapping from V to $[1..n]$). We sort the adjacency list (i.e. $OUT(x)$) of each vertex x in the increasing order of σ value. T is initialized as the ordered DFS tree of G with respect to these sorted adjacency lists. Throughout the sequence of edge deletions, our algorithm maintains the following invariant.

$\mathcal{I}$: At any instant of time, T is the ordered DFS tree corresponding to σ.

Consider the deletion of an edge (u, v). We first remove u from $IN(v)$ and v from $OUT(u)$. If (u, v) is not a tree edge, then this is all that is required. Otherwise we scan the vertices of $T(v)$ in the topological ordering, and process each vertex $x \in T(v)$ as follows.

1. If $IN(x)$ is empty, then we set $par(x)$ to null, and for each out-neighbor y of x, delete x from $IN(y)$.
2. If $IN(x)$ is non-empty then we hang x appropriately from its new parent in the DFS tree by executing procedure Hang(x) explained below.

Note : The processing of vertices of $T(v)$ in the topological ordering is very crucial because it ensures that before the processing of any vertex x begins, all its in-neighbors are already hung appropriately in the new DFS tree.

We now explain procedure Hang(x) (see the pseudocode on the next page). Let w be the vertex from $IN(x)$ with minimum finish time. Then the in-neighbors of x visited before w must lie on $path(w)$. Now consider any in-neighbor y of x lying on $path(w)$. Let z be its child on $path(w)$. Then at the time when DFS

traversal reaches w, vertex y would have scanned only upto those vertices in its adjacency list whose σ value is less than or equal to $\sigma(z)$. Thus if $\sigma(x) > \sigma(z)$, then x would not have been scanned by y and thus cannot be its child. But if $\sigma(x) < \sigma(z)$, then x would be a child of y in case it is unvisited at the time when DFS traversal reaches y. Based on this observation we can conclude the following. Let b be the first vertex in $IN(x)$ such that b is ancestor of w and the child of b on $path(w)$ has σ value greater than $\sigma(x)$ (see Figure 2). Then b is assigned as the parent of x. However, if no such vertex exists, then x is hung from w. Hanging x in this manner ensures that the invariant $\mathcal{I}$ is maintained.

Procedure Hang(x): hangs vertex x appropriately in T to preserve the invariant $\mathcal{I}$.

```
1  par(x) ← null;
2  w ← MinFinish(x);
3  for y ∈ IN(x)\{w} do
4  |  if y = LCA(y, w) then
5  |  |  z ← LA(w, 1+LEVEL(y));
6  |  |  if σ(x) < σ(z) then
7  |  |  |  par(x) ← y; break;
8  |  |  end
9  |  end
10 end
11 if par(x) = ∅ then par(x) ← w;
12 LEVEL(x) ← 1 + LEVEL(par(x));
13 Return;
```

Procedure MinFinish(x): computing x's in-neighbor having minimum finish time.

```
1  w ← First vertex in IN(x);
2  for y ∈ IN(x)\{w} do
3  |  z ← LCA(y, w);
4  |  if w = z then w ← y;
5  |  else if y ≠ z then
6  |  |  a ← LA(y, 1+LEVEL(z));
7  |  |  b ← LA(w, 1+LEVEL(z));
8  |  |  if σ(a) < σ(b) then
9  |  |  |  w ← y;
10 |  |  end
11 |  end
12 end
13 Return w;
```

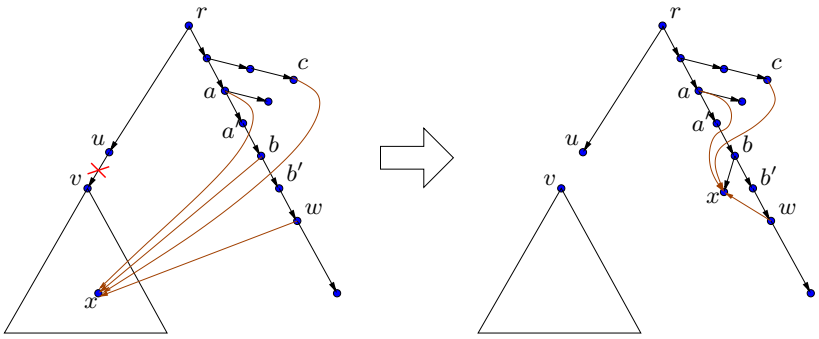

Fig. 2. Illustration of procedure Hang(x) : (i) w is vertex in $IN(x)$ having minimum finish time; (ii) x does not hang from c as it is not on $path(w)$; (iii) x does not hang from a as $\sigma(a') < \sigma(x)$; (iv) x hangs from b as $\sigma(x) < \sigma(b')$.

We now explain procedure MinFinish(x) that computes the in-neighbor w of x with minimum finish time. w is first initialized to be any arbitrary vertex

in $IN(x)$. Then we scan $IN(x)$, and for each $y \in IN(x)$ if FINISH-TIME$(y) <$ FINISH-TIME(w) then we update w to y. The FINISH-TIME of vertices y and w can be compared as follows. Let $z = LCA(y, w)$. If w is ancestor of y, i.e. $w = z$, then FINISH-TIME of w would be greater than that y, and vice-versa. Otherwise if there is no ancestor-child relationship between them, then let a, b be respectively the children of z on $path(y)$ and $path(w)$. Now FINISH-TIME of y would be less than that of w if and only if $\sigma(a) < \sigma(b)$. This is because a would be visited before b if $\sigma(a) < \sigma(b)$, and then START-TIME (as well as FINISH-TIME) of all the vertices in $T(a)$ would be less than that of vertices in $T(b)$, and vice versa.

We shall now analyze the time complexity of this decremental algorithm. The procedures Hang(x) and MinFinish(x) scan $IN(x)$ and use only LA/LCA queries that can be answered in $O(1)$ time (see Theorems 1 and 2). So, the time taken by each of them is $O(\deg(x))$. The scanning of the vertices of $T(v)$ in topological ordering can be done in linear time by using any integer sorting algorithm. Thus the time taken to handle deletion of tree edge (u, v) is $\Theta(\sum_{x \in T(v)} \deg(x))$. Now it follows from the discussion given above that a vertex x is processed during an edge deletion if that edge lies on the tree path from root to x. Suppose for a given sequence of edge deletions, this event happens $c(x)$ times for a vertex x. Then we get the following theorem.

Theorem 3. *There exists a decremental algorithm for maintaining a DFS tree T that takes $\Theta(\sum_x \deg(x) \cdot c(x))$ time where $c(x)$ is the number of times vertex x loses an edge on its path from the root in T during a given sequence of edge deletions.*

The algorithm for handling deletion of an edge described above is simple. However, this algorithm can be as inefficient asymptotically as recomputing the DFS tree from scratch after each edge deletion (see Observation 1 in Section 5).

4 An efficient randomized algorithm

The only difference between our deterministic algorithm described above and the randomized algorithm is that in the randomized version σ is chosen randomly uniformly in the beginning of the algorithm. We shall now show that for any arbitrary sequence of edge deletions, the algorithm will take expected $O(mn \log n)$ time to maintain a DFS tree.

4.1 Analysis of the algorithm

Consider vertices u and x such that x is reachable from u in G. x may be reachable from u by a direct edge or through a path from some of its outgoing neighbors. Let S_u be the set consisting of all those vertices $v \in OUT(u)$ such that at the time of deletion of edge (u, v), x was reachable from v. Note that x may also be present in set S_u if $(u, x) \in E$. It can be observed that for vertex $v \in OUT(u) \backslash S_u$, the deletion of (u, v) will have no influence on x, as at that time x was already unreachable from v. For each vertex $v \in S_u$, define DELETE-TIME(v)

as the time at which (u, v) is deleted. Let $\langle v_1, v_2, \ldots, v_k \rangle$ be the sequence of vertices from S_u arranged in increasing order of DELETE-TIME. Observe that every edge from $\{u\} \times S_u$, at the time of its deletion, could potentially be present on the path from root to x in T. However, we will now prove that if σ is selected randomly uniformly, then this may happen only $O(\log k)$ times on expectation for any given sequence of edge deletions. For this purpose, we first state a lemma that shows the relationship between any two vertices from set S_u. This lemma crucially exploits the fact the G is acyclic.

Lemma 2. *Suppose (u, v_i) is present in the DFS tree T. If there is any $j > i$ with $\sigma(v_j) < \sigma(v_i)$, then (i) START-TIME$(v_j)$ < START-TIME(v_i), and (ii) there is no path from v_j to v_i in the present graph.*

Proof (u, v_i) belongs to the DFS tree. So at the moment DFS visited u, v_i must be unvisited. Since $\sigma(v_j) < \sigma(v_i)$ and v_j has an incoming edge from u, so v_j will be visited (if not already visited) before v_i. So START-TIME(v_j) < START-TIME(v_i).

If there is a path from v_j to v_i in the present graph, then it follows from property **P2** of DFS traversal in DAG (stated in Lemma 1) that v_i must be a descendant of v_j in the DFS tree T. But this would imply that u is also descendant of v_j in T since u is parent of v_i in T. This tree path from v_j to u, if exists, and the edge $(u, v_j) \in E$ would form a cycle. This is a contradiction since the graph is acyclic. □

The following lemma precisely states the conditions under which the deletion of (u, v_i) will have no influence on x.

Lemma 3. *Suppose (u, v_i) is a tree edge at some time. If there is any $j > i$ with $\sigma(v_j) < \sigma(v_i)$, then x does not belong to $T(v_i)$ at that time.*

Proof As stated above vertex x is reachable from v_j, or x is vertex v_j itself. So from property **P1** stated in Lemma 1 it follows that FINISH-TIME(x) ≤ FINISH-TIME(v_j). It also follows from Lemma 2 that START-TIME(v_j) < START-TIME(v_i) and there is no path from v_j to v_i in the graph. Therefore, FINISH-TIME(v_j) < START-TIME(v_i). Hence we get the following relations.

$$\text{FINISH-TIME}(x) \leq \text{FINISH-TIME}(v_j) < \text{START-TIME}(v_i)$$

So v_i can not be ancestor of x in T. Hence $x \notin T(v_i)$. □

Lemma 3 leads to the following corollary.

Corollary 1. *If there exists j satisfying $i < j$ and $\sigma(v_j) < \sigma(v_i)$, then the deletion of the edge (u, v_i) will have no influence on the path from root to x in the DFS tree.*

Lemma 4. *On expectation, there are $O(\log k)$ edges from $\{(u, v_i) | 1 \leq i \leq k\}$ that may lie on the tree path from u to x at the time of their deletion.*

Proof For each vertex $v_i, 1 \leq i \leq k$, we define a random variable Y_i as follows. Y_i is 1 if x belongs to $T(v_i)$ at the time of deletion of edge (u, v_i), and 0 otherwise. Let $Y = \sum_{i=1}^{k} Y_i$. It follows from Corollary 1 that if $Y_i = 1$ then for all $j > i$, $\sigma(v_j)$ is greater than $\sigma(v_i)$. As σ is a uniformly random permutation, so

$$P[\sigma(v_j) > \sigma(v_i),\ \forall j,\ i < j \leq k] = \frac{1}{k-i+1}$$

Thus $E[Y_i] \leq \frac{1}{k-i+1}$, and so using linearity of expectation, $E[Y] \leq \sum_{i=1}^{k} \frac{1}{k-i+1} = O(\log k)$. □

Hence, if x is reachable from u in the initial graph, then for any arbitrary sequence of edge deletions, we can conclude the following. Upon deletion of expected $O(\log n)$ outgoing edges of u, x will have to be re-hung in the DFS tree. Therefore, using Theorem 3, the expected time complexity of the randomized algorithm for processing any arbitrary sequence of edge deletions is $O(mn \log n)$.

Theorem 4. *Let G be a DAG on n vertices and m edges and let $r \in V$. A DFS tree rooted at r can be maintained under deletion of edges using an algorithm that takes expected $O(mn \log n)$ time for any arbitrary sequence of edge deletions.*

5 Lower bounds for dynamic DFS tree problem

5.1 Maintaining ordered DFS tree explicitly

We show that the problem of maintaining the ordered DFS tree explicitly may require $\Omega(mn)$ time in total for directed as well as undirected graphs. For this we provide construction of graphs with $\Theta(n)$ vertices and then present a sequence of edge deletions and an ordering σ such that each edge deletion results in changing the parent of $\Theta(n)$ vertices in the ordered DFS tree defined by σ.

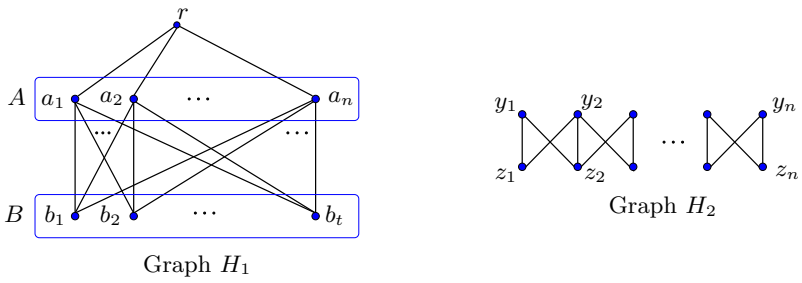

Fig. 3. Subgraphs H_1 and H_2

First we prove the result for the undirected graphs. Let $t = \lfloor m/n \rfloor$. Let H_1 be a graph with $n + t + 1$ vertices - $r, a_1, .., a_n, b_1, .., b_t$. The edges of H_1 are the pairs (r, a_i) and (a_i, b_j) for all i, j such that $1 \leq i \leq n$ and $1 \leq j \leq t$. Let H_2 be another graph with $2n$ vertices - $y_1, .., y_n, z_1, .., z_n$, such that each y_k has edges to z_{k-1}, z_k and z_{k+1} (see Figure 3). Now graph G is obtained by adding an edge between b_j and y_1 if j is odd, and between b_j and z_1 if j is even. This completes the construction of graph G. Let $\sigma =< r, y_1, .., y_n, z_1, .., z_n, a_1, .., a_n, b_1, .., b_t >$. We delete the edges in n stages. In stage i, we delete the edges incident on vertex a_i and the sequence of deletions is - $(a_i, b_1), .., (a_i, b_t)$.

Lemma 5. *Deletion of each edge (a_i, b_j) leads to the change of parent of all vertices in H_2 in the ordered DFS tree T defined by σ.*

Proof Consider the DFS tree just before the deletion of edge (a_i, b_j). The DFS traversal starts with r and visit vertices a_1 to a_i. After a_i it visits b_j, and then visits y_1 or z_1 depending on whether j is odd or even. If y_1 is the first vertex visited in H_2 then the sequence of vertices visited in H_2 would be $y_1, z_1, y_2, z_2, .., y_n, z_n$. And if z_1 is visited first then the sequence would be $z_1, y_1, z_2, y_2, .., z_n, y_n$. Thus if j is odd then for each k, y_k is parent of z_k in T, and if j is even then for each k, y_k is child of z_k in T. Hence each edge deletion (a_i, b_j) results in changing of parent of all vertices in H_2. □

We now prove the result for the directed graphs. Let H_1 be the same graph as described earlier except that (r, a_i) and (a_i, b_j) are now directed edges. Let H_3 be a graph on n vertices - $x_1, .., x_n$ without any edge. The directed graph G' is obtained by adding edges from each b_j to all the vertices in H_3. The ordering is $\sigma =< r, x_1, .., x_n,\ a_1, .., a_n, b_1, .., b_t >$ and we use the same sequence of edge deletions as for the undirected case.

Lemma 6. *Deletion of each edge (a_i, b_j) leads to the change of parent of all vertices in H_3 in the ordered DFS tree T defined by σ.*

Proof Note that at any instant of time all the vertices of H_3 would be hanging from the same vertex in H_1. Now after deletion of (a_i, b_j) edge, each vertex of H_3 will hang from vertex b_{j+1} if $j < n$, and b_1 otherwise. Thus on each edge deletion, the parent of each vertex in H_3 changes. □

Theorem 5. *For each pair of positive integers n, m ($m > 3n$), there exists a directed (undirected) graph with $\Theta(n)$ vertices and $\Theta(m)$ edges such that any decremental algorithm for maintaining the ordered DFS tree explicitly in this graph requires $\Omega(mn)$ time.*

Observation 1 *In the directed graph G' described above, each vertex x_k in H_3 changes its parent on deletion of (a_i, b_j) edge. So the total number of times x_k changes its parent is $\Theta(m)$. Notice that* $\deg(x_k)$ *remains $t = \lfloor m/n \rfloor$ throughout the sequence of edge deletions. Thus Theorem 3 implies that the total time taken by the deterministic algorithm in Section 3 for this graph is $\Theta(m^2)$.*

5.2 Partial dynamic ordered DFS tree and static all-pairs reachability

Let $G = (V, E)$ be a directed graph. Let $\mathcal{R}(v)$ denote the set of vertices reachable from $v \in V$. The objective of the all-pairs reachability problem is to compute $\mathcal{R}(v)$ for each $v \in V$. Let $\mathcal{A}$ be any decremental algorithm for maintaining the ordered DFS tree from a vertex in a directed graph. We shall now show that we can compute $\mathcal{R}(v)$ for all v by executing $\mathcal{A}$ on a graph G' formed by a suitable augmentation of G.

Add a dummy vertex s to G and connect it to each vertex of G by an edge. This completes the description of graph G'. Let $v_1, \ldots, v_n$ be any arbitrary

sequence of V and let σ be the ordering defined by this sequence. We sort all the edges in the adjacency lists of G' according to σ. This takes $O(m)$ time using any integer sorting algorithm.

We now execute $\mathcal{A}$ on graph G' to maintain the ordered DFS tree from s. The sequence of edge deletions is $(s, v_1), (s, v_2), ..., (s, v_{n-1})$. The definition of ordered DFS tree implies the following. Just before the deletion of edge (s, v_i), the set of vertices in the subtree $T(v_i)$ is exactly equal to $R(v_i)$. So we can compute $R(v_i)$ by traversing $T(v_i)$ just before the deletion of edge (s, v_i). In this manner, at the end of deletion of edge (s, v_{n-1}), we have obtained entire reachability information of the graph. Since $\sum_i |R(v_i)| = O(n^2)$, we can state the following theorem.

Theorem 6. *Let $f(m, n)$ be the total time taken by any decremental algorithm for maintaining the ordered DFS tree in a directed graph for any sequence of n edge deletions. If the algorithm can report DFS tree at any stage in $O(n)$ time, we can compute all-pairs reachability of a directed graph on n vertices and m edges in $O(f(m, n) + n^2)$ time.*

Static all-pairs reachability is a classical problem and its time complexity is $O(\min(mn, n^\omega))$ [16]. This bound has remained unbeaten till now. So it is natural to believe that $O(\min(mn, n^\omega))$ is the best possible bound on the time complexity of the static all-pairs reachability. Therefore, Theorem 6 implies the following conditional lower bounds on the dynamic complexity of ordered DFS tree.

Theorem 7. *Let G be a directed graph on n vertices and m edges. Let $\mathcal{A}$ be a decremental algorithm for maintaining the ordered DFS tree in G with $O(n)$ query time. The following lower bounds hold for $\mathcal{A}$ unless we have an $o(\min(mn, n^\omega))$ time algorithm for the static all-pairs reachability problem.*

- *The total update time for any sequence of edge deletions is $\Omega(\min(mn, n^\omega))$.*
- *The worst case time to handle an edge deletion is $\Omega(\min(m, n^{\omega-1}))$.*

For an algorithm that takes more than $O(n)$ query time our reduction implies the following lower bound.

Theorem 8. *Let $q(m, n)$ be the worst case query time to report the DFS tree and $u(m, n)$ be the worst case time to handle an edge deletion by a decremental algorithm for maintaining the ordered DFS tree. Then either $q(m, n)$ or $u(m, n)$ must be $\Omega(\min(m, n^{\omega-1}))$ unless we have an $o(\min(mn, n^\omega))$ time algorithm for the static all-pairs reachability problem.*

Remark 1. We can obtain exactly same conditional lower bounds as in Theorem 7 and Theorem 8 for any incremental algorithm for the ordered DFS tree in a digraph. The graph G' is the same except that we insert the edges in the order $(s, v_n), \ldots, (s, v_1)$, and traverse the subtree $T(v_i)$ in the DFS tree just after the insertion of (s, v_i).

6 Conclusion

We presented the first decremental algorithm for maintaining a DFS tree in a DAG. This decremental algorithm as well as the incremental algorithm of Franciosa [6] crucially exploit the acyclic condition of a DAG in order to achieve efficient update time. In fact it can be shown that there exists a directed graph on which these algorithms perform no better than rebuilding the DFS tree form scratch after each update (see the full version of this article for the details). So we would like to conclude with the following open question whose answer will surely add significantly to our understanding of the dynamic DFS tree problem.

– Does there exist an incremental/decremental algorithm with $O(mn)$ update time for DFS tree in general directed graphs ?

References

1. Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *ICALP*, pages 73–84, 2000.
2. Surender Baswana and Shahbaz Khan. Incremental algorithm for maintaining DFS tree for undirected graphs. In *ICALP (1)*, pages 138–149, 2014.
3. Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.
4. Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
5. Paolo Giulio Franciosa, Daniele Frigioni, and Roberto Giaccio. Semi-dynamic breadth-first search in digraphs. *Theor. Comput. Sci.*, 250(1-2):201–217, 2001.
6. 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.
7. 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.
8. Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994.
9. John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.
10. John H. Reif. A topological approach to dynamic graph connectivity. *Inf. Process. Lett.*, 25(1):65–70, 1987.
11. Liam Roditty and Uri Zwick. On dynamic shortest paths problems. In *ESA*, pages 580–591, 2004.
12. Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.
13. Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012.
14. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
15. Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.
16. Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, pages 887–898, 2012.