On Dynamic DFS Tree in Directed Graphs

Surender Baswana and Keerti Choudhary

Department of CSE, IIT Kanpur
Kanpur, 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* may require $\Omega(n^3)$ 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 [18], 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 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 [5, 10, 15, 16, 19].

In this paper we address the problem of maintaining a DFS tree in a dynamic graph. We believe that efficient algorithms for this problem will have potential to provide efficient algorithms for many other graph problems. One such problem is maintaining topological ordering in a directed acyclic graph (DAG). Though there are efficient incremental algorithms for this problem [3, 9], there is no nontrivial algorithm for topological ordering in a fully dynamic environment. Observe that we just need to compare the finish time of two vertices in the DFS tree to determine their order in the topological ordering. Using standard data structures for dynamic trees [17], this information can be retrieved in $O(\log n)$ time. Hence an efficient dynamic algorithm for DFS tree in a DAG will immediately imply a fully dynamic algorithm for topological ordering.

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 [12, 13] and Milterson et al. [11] addressed the complexity of the ordered DFS problem in a dynamic environment. Milterson et al. [11] 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. [7, 8] 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. [6] designed a decremental algorithm that achieves $O(n)$ amortized time per edge deletion.

In this paper, 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 Decremental algorithm for DFS tree in DAG

We present a decremental algorithm for maintaining a DFS tree in a DAG that takes 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 our 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 for a vertex in a DFS tree depends upon the order in which its in-neighbors get visited during the DFS traversal.

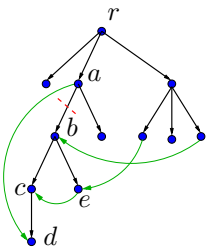

Fig. 1. Deletion of edge (a, b) results in the change of parent of every vertex in $T(b)$.

In order to achieve better 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 $\Omega(n^3)$ 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 maintaining a DFS tree explicitly is provably faster than maintaining an ordered DFS tree by a factor of $\Omega(n)$ in the incremental environment.

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 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 [14] for decremental (or incremental) BFS tree problem. This reduction implies a 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 $\omega < 2.373$ [20].

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 $\Theta(n^4)$ time on a graph with $\Theta(n^2)$ edges. 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 : DFS tree of G rooted at r at any particular time.
- 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 DFS traversal is finished 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 topological order.
- $OUT(x)$: The list of vertices of T having an incoming edge from x.
- $par(x)$: Parent of x in T.
- $path(x)$: Path from r to x in T.
- LEVEL(x) : Level of a vertex x in T s.t. 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[4]). *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 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 proof in the Appendix) 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. If there is a path from y to z, then the following two properties hold true for each DFS traversal carried out in the graph.*

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.

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

Consider deletion of an edge (u, v). We first remove u from $IN(v)$ and v from $OUT(u)$. Now if (u, v) is not a tree edge, then we just delete edge (u, v). 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 its parent pointer 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 that the processing of vertices in the topological ordering is helpful because when a vertex x is processed the positions of all its in-neighbors would have already been fixed in the new DFS tree.

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

```
1  par(x) ← ∅;
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;
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;
```

We now explain procedure Hang(x). 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 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 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.

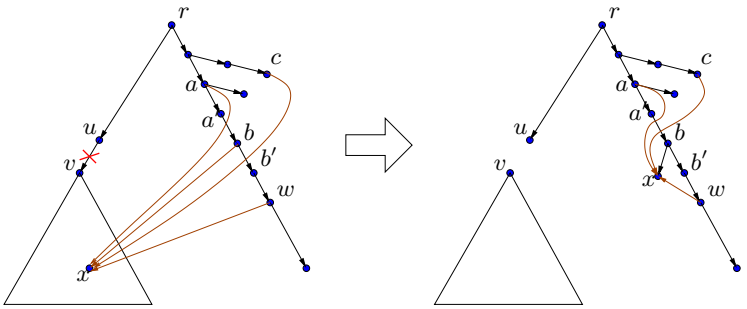

Fig. 2. Illustration of procedure Hang(x) : (i) w is vertex in $IN(x)$ having minimum finish time; (ii) x does not hang from c as it is not on $path(w)$; (iii) x does not hang from a as $\sigma(a') < \sigma(x)$; (iv) x hangs from b as $\sigma(x) < \sigma(b')$.

We now explain procedure MinFinish(x) that computes the in-neighbor w of x with minimum finish time. w is first initialized to be an 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 procedure Hang(x) and MinFinish(x) scans $IN(x)$ and uses only LA/LCA queries that can be answered in $O(1)$ time (see Theorems 1 and 2). So, 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 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

Let x be any vertex 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 x to root 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*

1. START-TIME(v_j) < START-TIME(v_i).
2. *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$(x) \leq$ 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 $\Omega(n^3)$ time in total for directed as well as undirected graphs. For this we provide constructions 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.

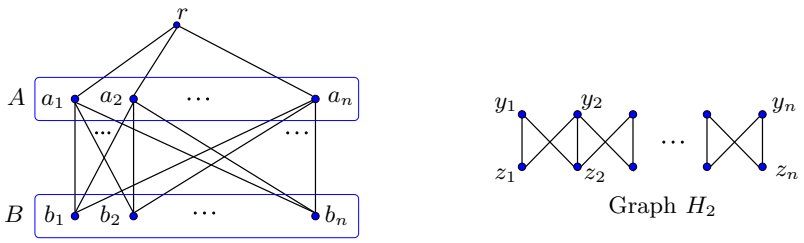

Fig. 3. Subgraphs H_1 and H_2

First we prove the result for the undirected graphs. Let H_1 be a graph with $2n + 1$ vertices - $r, a_1, .., a_n, b_1, .., b_n$. The edges of H_1 are the pairs (r, a_i) and (a_i, b_j) for i, j in range 1 to n. Let H_2 be another graph with $2n$ vertices -

$y_1, .., y_n, z_1, .., z_n$, such that each y_t has edges to z_{t-1}, z_t and z_{t+1}. (see Figure 3). Now graph G is obtained by adding edge from b_j to y_1 whenever j is odd, and from b_j to z_1 when j is even. This completes the construction of graph G. Let $\sigma =< r, y_1, .., y_n, z_1, .., z_n, a_1, .., a_n, b_1, .., b_n >$. We delete the edges in stages. In stage i, we delete the edges incident on vertex a_i and the sequence of deletions is - $(a_i, b_1), .., (a_i, b_n)$.

Lemma 5. *On each edge deletion (a_i, b_j), parent of all the vertices in H_2 is altered 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/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 t, y_t is parent of z_t in T, and if j is even then for each t, y_t is child of z_t 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 - $x_1, .., x_n$ where each x_t is a singleton vertex. Directed graph G' is obtained by adding edges from each b_j to all vertices in H_3. The ordering is $\sigma =< r, x_1, .., x_n,\ a_1, .., a_n, b_1, .., b_n >$ and we use the same sequence of edge deletions as for the undirected case.

Lemma 6. *On each edge deletion (a_i, b_j), parent of all the vertices in H_3 is altered 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, parent of all vertices in H_3 changes. □

Theorem 5. *For each $n > 1$, there exists a directed (undirected) graph with $\Theta(n)$ vertices such that any decremental algorithm for maintaining the explicit ordered DFS tree for it requires $\Omega(n^3)$ time in total.*

Observation 1 *In the directed graph G' described above, each vertex x_t in H_3 changes its parent on deletion of (a_i, b_j) edge. So the total number of times x_t changes its parent is $\Theta(n^2)$. Notice that the in-degree of x_t remains n through out the sequence of edge deletions. Thus Theorem 3 implies that the total time taken by the deterministic algorithm in Section 3 for this graph will be $\Theta(n^4)$.*

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' of almost same size formed by suitable augmentation of G.

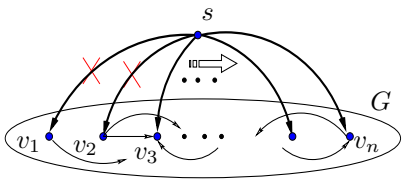

Fig. 4. Graph G'

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. Figure 4 illustrates G'.

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 $O(\min(mn, n^\omega))$ [20]. 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 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 deletion of edges 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 Theorems 7, 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 DFS tree in a DAG and also provided conditional lower bounds for the hardness of ordered DFS tree problem in dynamic setting. Our decremental algorithm crucially uses the acyclic condition of a DAG in order to achieve $O(mn \log n)$ time complexity. In fact we can show that there exists a directed graph on which this algorithm may take $\Omega(n^4)$ time on expectation (see Appendix). So we would like to conclude with the following open question whose answer will surely add significantly to our understanding about 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. Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. A new approach to incremental topological ordering. In *SODA*, pages 1108–1115, 2009.
4. Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.
5. Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
6. Paolo Giulio Franciosa, Daniele Frigioni, and Roberto Giaccio. Semi-dynamic breadth-first search in digraphs. *Theor. Comput. Sci.*, 250(1-2):201–217, 2001.
7. Paolo Giulio Franciosa, Giorgio Gambosi, and Umberto Nanni. On the structure of dfs-forests on directed graphs and the dynamic maintenance of dfs on dag's. In *ESA*, pages 343–353, 1994.
8. 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.
9. Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms*, 8(1):3, 2012.

10. 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.
11. Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994.
12. John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.
13. John H. Reif. A topological approach to dynamic graph connectivity. *Inf. Process. Lett.*, 25(1):65–70, 1987.
14. Liam Roditty and Uri Zwick. On dynamic shortest paths problems. In *ESA*, pages 580–591, 2004.
15. Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.
16. Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012.
17. Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.
18. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
19. Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.
20. Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012.

A Appendix

A.1 Proof of Lemma 1

Let S be the set of vertices (including y and z) that lie on some path from y to z. Let s be the vertex from set S that is visited first during a DFS traversal. Note that z will be in the subtree rooted at s. So we will have -

$$\text{START-TIME}(s) \leq \text{START-TIME}(z) < \text{FINISH-TIME}(z) \leq \text{FINISH-TIME}(s) \quad (1)$$

We consider two cases -

1. If $s = y$, then z will be descendant of y, and we will have $\text{FINISH-TIME}(z) < \text{FINISH-TIME}(y)$. Hence both **P1** and **P2** are true in this case.
2. Now if $s \neq y$, then by the definition of s, vertex s is visited before y. Now as y cannot be descendant of s, there cannot be any ancestor-child relationship between s and y. (Note that this is not true for general directed graphs). Using these two facts we have -

   $$\text{START-TIME}(s) < \text{FINISH-TIME}(s) < \text{START-TIME}(y) < \text{FINISH-TIME}(y) \quad (2)$$

   It follows from Inequalities 1 and 2 that $\text{START-TIME}(z) < \text{START-TIME}(y)$ and $\text{FINISH-TIME}(z) < \text{FINISH-TIME}(y)$. So in this case also **P1** holds true, and as $\text{START-TIME}(z) < \text{START-TIME}(y)$, **P2** is vacuously true.

A.2 The efficiency of the decremental algorithm for DFS tree on a directed graph

The incremental algorithm of Franciosa et al. [7, 8] and the decremental algorithm presented in this paper work only for DAG. It is natural to think how critical is the acyclic condition of the DAG for the efficiency of these algorithms. Indeed, it is very critical. To show this, we shall construct a directed graph G on n vertices and prove that there exists a sequence of edge insertions for which the incremental algorithm of Franciosa et al. [7, 8] takes $\Theta(n^4)$ time. Then we shall slightly modify the graph G and present a sequence of edge deletions for which our decremental algorithm takes $\Theta(n^4)$ time on expectation.

We begin with the following Lemma that captures the subtle differences between a DFS tree for general directed graph and a DFS tree for DAG.

Lemma 7. *Let T be any DFS tree for G and let u and x be vertices such that* FINISH-TIME*$(x)>$*FINISH-TIME*(u). Then*

- *if G is acyclic, then there can not exist any path from u to x in G.*
- *if G is a general directed graph, then a path from u to x, if exists, must contain some ancestor of u.*

We shall use the following terminology for both the algorithms. Upon any edge update, we say that a vertex x changes its position in the DFS tree if the path from root to x in the DFS tree gets modified. A fact that holds for both the algorithms is that whenever x changes its position, it incurs a computation cost of $deg(x)$.

Worst case example for the Incremental algorithm

The algorithm of Franciosa et al. [7, 8] takes $O(mn)$ time for maintaining a DFS tree in a DAG for any sequence of m edge insertions. Whenever an edge (u, v) is inserted, the vertex v is appended to the end of the adjacency list of u. At each stage, the algorithm maintains the ordered DFS tree according to these adjacency lists. Since there is no cycle, it follows from Lemma 7 that a vertex x changes its position in the DFS tree upon insertion of an edge (u, v) if and only if x was not reachable from u before the insertion of (u, v) and has become reachable from u after the insertion of (u, v). This implies that insertion of at most one outgoing edge of u can change the position of x in the DFS tree. Hence vertex x can charge $deg(x)$ cost to any vertex $u \in V$ only once during the algorithm. This leads to $O(mn)$ time complexity of the incremental algorithm. We shall now provide construction of a directed graph G on $5n + 3$ vertices. This graph has two subsets of vertices U and X, each of size n. We shall present a sequence of edge insertions such that whenever an outgoing edge is added to $u \in U$, all vertices in X change their position in the DFS tree. Out-degree of each vertex in X will be $n - 1$ and this will imply $\Theta(n^4)$ update time.

Description of graph G:
The graph G consists of two subgraphs H_0 and H_1 (see Figure 5). Subgraph H_0 is a tree of height one rooted at r with $a_1, a_2, \ldots, a_n$ as its leaf nodes. Subgraph H_1 is a directed graph containing cycles and it consists of the following four parts.

1. A subgraph H_0' which is a tree of height one rooted at s with $b_1, b_2, \ldots, b_n$ as its leaf nodes.
2. A directed chain $C = (v_n, v_{n-1}, \ldots, v_1, v_0)$. For $j = 1$ to n, (v_j, s) is an edge in H_1.
3. A set $X = \{x_1, .., x_n\}$ of n vertices. The subgraph induced on X is a complete graph. For $j = 1$ to n, (v_0, x_j) is an edge in H_1.
4. A set $U = \{u_1, u_2, \ldots, u_n\}$ of n vertices. For $j = 1$ to n, (b_j, u_j) and (u_j, v_n) are edges in H_1. (Edges in the set $U \times \{v_n\}$ are not shown in the figure.)

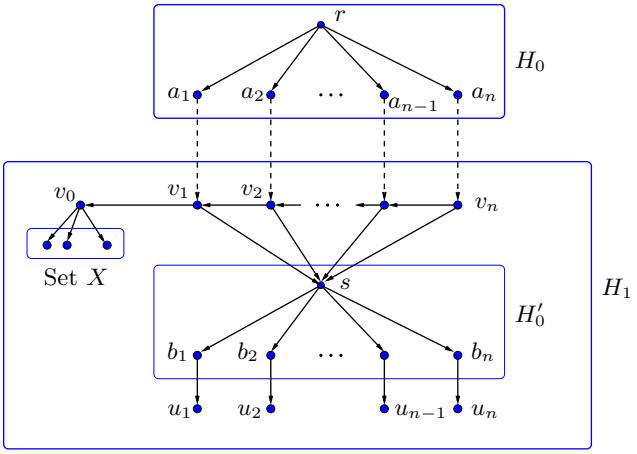

Fig. 5. Subgraphs H_0, H_1 of the directed graph G. Here H_0 is acyclic, whereas H_1 has many cycles.

We assume the following conditions for the order in which vertices appear in the adjacency lists in G.

C1 : $Adj(r) = (a_1, a_2, \ldots, a_n)$, i.e. a_j appears before a_{j+1} in adjacency list of r.
C2 : $Adj(s) = (b_1, b_2, \ldots, b_n)$, i.e. b_j appears before b_{j+1} in adjacency list of s.
C3 : For $j = 1$ to n, s is the first vertex in the adjacency list of v_j.

Note that the assumption stated above is indeed a valid assumption. This is because in the algorithm of Franciosa et al. [7, 8], whenever an edge (a, b) is inserted, the vertex b is appended at the end of the adjacency list of a. So we can impose any order on the adjacency lists of G and while constructing G incrementally, add the outgoing edges to each vertex according to that order only.

Sequence of Edge insertions:

The edges are inserted in n rounds : Rn to $R1$ (so here the first round is Rn and the last round is $R1$). In the round Rj we proceed as follows. First we insert the edge (a_j, v_j) between the subgraphs H_0 and H_1. Next we add the edges $U \times \{v_{j-1}\}$ in the order - (u_n, v_{j-1}), (u_{n-1}, v_{j-1}), ..., (u_1, v_{j-1}). Thus there are $(n+1)$ edges insertions in each round and there are total n rounds.

Analysis

Consider the round Rj. We will show that each edge insertion in this round will lead to a change in the position of all vertices of set X.

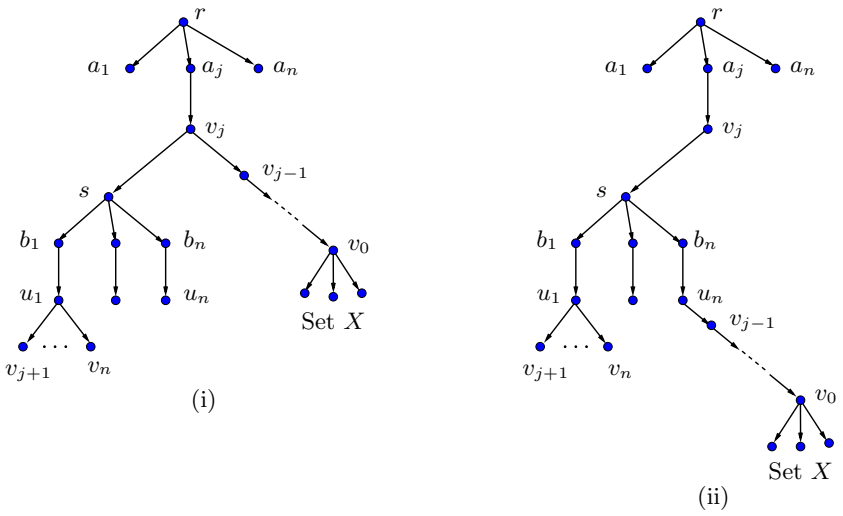

Fig. 6. DFS tree obtained after addition of (i) edge (a_j, v_j) (ii) edge (u_n, v_{j-1}).

Note that before the beginning of round Rj all the vertices in set U already have edges to the vertices $v_j, v_{j+1}, \ldots, v_n$. So from each $u_i \in U$ there exists a path to set X, but all such paths pass through vertex v_j.

Consider the insertion of edge (a_j, v_j) in round Rj. So till now the edges inserted between the subgraphs H_0 and H_1 are $(a_j, v_j), \ldots, (a_n, v_n)$. Since a_j is first vertex in $adj(r)$ which has an edge to subgraph H_1, the subgraph H_1 after insertion of edge (a_j, v_j) in round Rj would be hanging from vertex v_j. The DFS traversal after reaching vertex v_j will first visit s due to condition C3. The DFS tree after insertion of edge (a_j, v_j) is shown in Figure 6(i). After reaching s, vertex b_1 will be visited followed by u_1. Note that vertices $v_0, \ldots, v_n$ are reachable from u_1, but the paths to $v_0, \ldots, v_{j-1}$ are *blocked* by ancestor v_j. Hence only $v_{j+1}, \ldots, v_n$ will hang from u_1. The vertex v_{j-1} thus hangs from v_j only. Consider the entire subtree consisting of the path $(v_{j-1}, \ldots, v_0)$ with X hanging from v_0. When we insert the edge (u_n, v_{j-1}), this entire subtree will hang from u_n as shown in Figure 6(ii). When we insert the edge (u_{n-1}, v_{j-1}), this entire subtree will hang from u_{n-1}. In this way, the vertices of this subtree will change their position in the DFS tree total $n+1$ times in round Rj. This

shows that whenever an outgoing edge is added to $u \in U$, all vertices in X change their position in the DFS tree. Therefore, based on the discussion above, the incremental algorithm of Franciosa et al. [7, 8] for DFS tree will incur $\Theta(n^4)$ time on graph G.

Theorem 9. *There exists a graph on $\Theta(n)$ vertices and a sequence of $\Theta(n^2)$ edge insertions such that the algorithm of Franciosa et al. [7, 8] for maintaining a DFS tree takes $\Theta(n^4)$ time.*

Worst case example for the Decremental algorithm

We first provide an intuition for the limitation of our algorithm. The main reason for $O(mn \log n)$ time complexity of our decremental algorithm was the following. For a pair of vertices u and x, for any sequence of deletion of the outgoing edges of u, vertex x will be change its position in the DFS tree only expected $O(\log n)$ number of times. For the graph G_f that we will construct below, it will be shown that this condition no longer holds.

Let G_f be the final graph obtained after all edge insertions in the graph considered in previous subsection. Now suppose G_f is the input graph for the decremental setting, and the sequence of edge deletions are the same as sequence of edge insertions, but in the reverse order. If all the conditions C1, C2, and C3 are met for the adjacency lists of graph G_f, then on deletion of each edge, the set X would be changing its relative position in the DFS tree. This can be easily seen since the sequence of DFS trees obtained during edge insertions would now be just reversed in the deletions case.

But the adjacency lists for decremental case are randomly uniformly permuted. So none of the three conditions can be guaranteed. We reformulate these conditions as follows.

$C_{1,j}$: The vertex from the set $\{a_j, a_{j+1}, \ldots, a_n\}$ that is visited first during the DFS traversal starting from r is a_j.
$C_{2,j}$: The vertex from the set $\{b_j, b_{j+1}, \ldots, b_n\}$ that is visited first during the DFS traversal starting from s is b_j.
$C_{3,j}$: s is the first vertex in the Adjacency list of v_j.

Suppose we take a uniformly random permutation σ of vertices and sort the Adjacency list of each vertex according to it. Then condition $C_{3,j}$ will hold for $n/2$ values of j on expectation. In order to take care of conditions $C_{1,j}$ and $C_{2,j}$, we modify the subgraphs H_0 and H'_0 as shown in Figure 7. It can be observed that, in the modified graph H_0, with probability half just before deletion of (a_j, v_j), the subgraph H_1 is hanging from vertex v_j.

Sequence of Edge deletions
The edges are deleted in n rounds : $R1$ to Rn. In the round Rj we proceed as follows. First we delete the edges from set $U \times \{v_{j-1}\}$ in the order - (u_1, v_{j-1}), (u_2, v_{j-1}), ..., (u_n, v_{j-1}). Next we delete the edge (a_j, v_j).

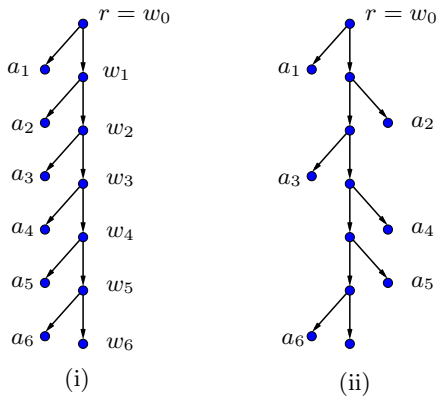

Fig. 7. (i) Directed acyclic graph H_0 for the decremental case. (ii) A DFS tree obtained for graph H_0 when adjacency lists are uniformly randomly permuted.

Analysis

For the purpose of analysis, we define a subset of rounds $\mathcal{R}$ and a subset of vertices $U' \subseteq U$.

- $\mathcal{R} = \{Rj \mid \text{Conditions } C_{1,j} \text{ and } C_{3,j} \text{ hold}\}$.
- $U' = \{u_i \mid \text{Condition } C_{2,i} \text{ holds}\}$.

Consider a round $Rj \in \mathcal{R}$. We will have that the subgraph H_1 hangs from vertex v_j, and the DFS traversal after reaching v_j first visits vertex s. Consider the stage when the edges in $\{u_1, \ldots, u_{i-1}\} \times \{v_{j-1}\}$ have already been deleted. The next edge to be deleted is (u_i, v_{j-1}). If the condition $C_{2,i}$ holds, then the vertices $v_{j-1}, \ldots, v_0$ and set X will hang from u_i just before the deletion of (u_i, v_{j-1}). So during round Rj, the deletion of an outgoing edge from each $u \in U'$ results in the change in position of all vertices in set X.

Thus the vertices in set X change their position at least $|\mathcal{R}| \cdot |U'|$ times. Recall that the subgraph induced by X is a complete graph. Hence the running time of the decremental algorithm is of the order of $n^2 \cdot |\mathcal{R}| \cdot |U'|$. It can be observed that the expected size of $\mathcal{R}$ is $n/4$, and the expected size of U' is $n/2$. The random variables $|\mathcal{R}|$ and $|U'|$ are independent. Therefore, the expected running time of the decremental algorithm on G_f will be $\Theta(n^4)$.

Theorem 10. *There exists a graph on $\Theta(n)$ vertices, $\Theta(n^2)$ edges, and a sequence of edge deletions such that our decremental algorithm for maintaining a DFS tree takes $\Theta(n^4)$ time on expectation.*