Incremental Algorithm for Maintaining DFS Tree for Undirected Graphs

Surender Baswana* and Shahbaz Khan**

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

Abstract. Depth First Search (DFS) tree is a fundamental data structure for graphs used in solving various algorithmic problems. However, very few results are known for maintaining DFS tree in a dynamic environment - insertion or deletion of edges. The only non-trivial result for this problem is by Franciosa et al. [4]. They showed that, for a directed acyclic graph on n vertices, a DFS tree can be maintained in $O(n)$ amortized time per edge insertion. They stated it as an open problem to maintain a DFS tree dynamically in an undirected graph or general directed graph.
We present the first algorithm for maintaining a DFS tree for an undirected graph under insertion of edges. For processing any arbitrary online sequence of edge insertions, this algorithm takes total $O(n^2)$ time.

Keywords: dynamic·incremental·undirected graph·depth first search

1 Introduction

Depth First Search (DFS) is a well known graph traversal technique. This technique has been reported to be introduced by Charles Pierre Trémaux, a 19th-century French mathematician who used it for solving mazes. However, it was Tarjan, who in his seminal work [9], demonstrated the power of DFS traversal for solving various fundamental graph problems, namely, topological sorting, connected components, biconnected components, strongly-connected components, etc.

DFS traversal is a recursive algorithm to traverse a graph. This traversal produces a rooted spanning tree (or forest), 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).

A DFS tree, say T, imposes the following relation on each non-tree edge $(x, y) \in E \backslash T$.

* 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.

$\mathcal{R}(x, y)$: Either x is an ancestor of y in T or y is an ancestor of x in T.

This elegant relation defined by a DFS tree has played the key role in solving various graph problems. Similar relations exists for the case of DFS tree in directed graphs.

Most of the graph applications in real world deal with graphs that keep changing with time. An algorithmic graph problem is modeled in the dynamic environment as an online sequence of insertion and deletion of edges. The aim is to maintain the solution of the given problem after each edge 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. A dynamic algorithm is called an incremental algorithm if it supports only insertion of edges.

In spite of the simplicity and elegance of a DFS tree, its parallel and dynamic versions have turned out to be quite challenging. In fact, in the dynamic setting, the ordered DFS problem (where the edges are visited strictly in the order given by the adjacency list of the graph) is shown to be hard by Reif[6, 7]. He showed that ordered DFS problem is a P-Complete problem. Milterson et al. [5] later proved that if dynamic version of any non-redundant P-Complete problem is updatable in $t(n)$ time, then every problem in P is updatable in $O(t(n) + \mathbf{polylog}(n))$ time. So it is highly unlikely that any $O(\mathbf{polylog}(n))$ update time algorithm exists for the ordered DFS problem. Though the ordered DFS problem is significant from the perspective of complexity theory, none of the existing algorithmic applications of DFS trees require such restrictions. Hence it is natural to address the problem of maintenance of any DFS tree in a dynamic graph.

For the case of directed acyclic graphs, Franciosa et al. [4] presented an incremental algorithm for maintaining DFS tree in $O(mn)$ total time. This is the only non-trivial result available for the dynamic DFS tree problem. Maintaining DFS tree incrementally for undirected graph (or general directed graph) was stated as an open problem by Franciosa et al. [4]. The following short discussion may help one realize the non-triviality of maintaining a DFS tree incrementally in an undirected graph.

Consider insertion of an edge (x, y). If $\mathcal{R}(x, y)$ holds, then no change in DFS tree is required. Such an edge is called a *back edge*. Otherwise, the relation $\mathcal{R}(x, y)$ does not hold for edge (x, y), and we call such an edge a *cross* edge. See Figure 1 for a better visual description. Let w be the lowest common ancestor of x and y. Let u and v be its children such that $x \in T(u)$ and $y \in T(v)$. Insertion of (x, y) violates the property of a DFS tree as follows.

Let S be the set of visited vertices when the DFS traversal reached w. Since $T(u)$ and $T(v)$ are two disjoint subtrees hanging from w, the vertices of $T(u)$ and $T(v)$ belong to disjoint connected components in the subgraph induced by $V \backslash S$. Insertion of edge (x, y) connects these components such that the vertices of $T(u) \cup T(v)$ have to hang as a single subtree from w. This implies that $T(u)$ will have to be rerooted at x and hung from y (or $T(v)$ has to be rerooted at y and hung from x). This rerooting will force restructuring of $T(u)$ because, in order to keep it as a DFS subtree, we need to preserve the relation $\mathcal{R}$ for every non-tree

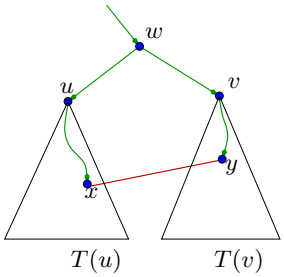

Fig. 1. (x, y) is a cross edge.

edges in $T(u)$. Observe that it is not obvious to perform this restructuring in an efficient manner.

We present the first incremental algorithm for maintaining a DFS tree (or DFS forest if the graph is not connected) in an undirected graph. Our algorithm takes a total of $O(n^2)$ time to process any arbitrary online sequence of edges. Hence the amortized update time per edge insertion is $\Omega(n^2/m)$, which is $O(1)$ for the case $m = \Omega(n^2)$. In addition to the $O(m + n)$ space occupied by the graph, our algorithm uses only $O(m + n)$ extra space. Moreover, excluding the standard data structures for ancestors in a rooted tree [1, 2], our algorithm employs very simple data structures. These salient features make this algorithm an ideal candidate for practical applications.

1.1 Related Work

Breadth first search (BFS) is another popular graph traversal algorithm. A BFS tree can be maintained incrementally in $O(mn)$ time [3], improving which is shown to be hard [8].

2 Preliminaries

Given an undirected graph $G = (V, E)$ on $n = |V|$ vertices and $m = |E|$ edges, the following notations will be used throughout the paper.

- T : A DFS tree of G at any particular time.
- r : Root of tree T.
- $par(v)$: Parent of v in T.
- $P(u, v)$: Path between u and v in T.
- $\textsc{level}(v)$: Level of a vertex v in T s.t. $\textsc{level}(r) = 0$ and $\textsc{level}(v) = \textsc{level}(par(v)) + 1$.
- $\textsc{level}(e)$: Level of an edge $e = (x, y)$ in T s.t. $\textsc{level}(e) = \min(\textsc{level}(x), \textsc{level}(y))$.
- $T(x)$: The subtree of T rooted at a vertex x.
- $LCA(u, v)$: The Lowest Common Ancestor of u and v in tree T.
- $LA(u, k)$: The ancestor of u at level k in tree T.

We explicitly maintain the level of each vertex during the algorithm. Since the tree grows from the root downward, a vertex u is said to be at higher level than vertex v if $\text{LEVEL}(u) < \text{LEVEL}(v)$. Similar notion is used for edges.

The DFS tree T maintains the following data structure at each stage.

- Each vertex v keeps a pointer to $par(v)$ and $par(r) = r$.
- T is also stored as an adjacency list for traversing $T(v)$ from v.
- Each vertex v keeps a list $\mathcal{B}(v)$ which consists of all those back edges that originate from $T(v)$ and terminate at $par(v)$. This, apparently uncommon and perhaps unintuitive, way of keeping the back edges leads to efficient implementation of the rerooting procedure.

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[2]). *There exists a dynamic data structure for a rooted tree T that uses linear space and can report $LCA(x, y)$ in $O(1)$ time for any two vertices $x, y \in T$. The data structure supports insertion or deletion of any leaf node in $O(1)$ time.*

Theorem 2 (Alstrup and Holm 2000[1]). *There exists a dynamic data structure for a rooted tree T that uses linear space and can report $LA(u, k)$ in $O(1)$ time for any vertex $u \in T$. The data structure supports insertion of any 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.

If the graph is not connected, the aim would be to maintain a DFS tree for each connected component. However, our algorithm, at each stage, maintains a single DFS tree which stores the entire forest of these DFS trees as follows. We add a dummy vertex s to the graph in the beginning and connect it to all the vertices. We maintain a DFS tree of this augmented graph rooted at s. It can be easily seen that the subtrees rooted at children of s correspond to DFS trees of various connected components of the original graph.

3 Overview of the algorithm

Our algorithm is based on two principles. The first principle, called *monotonic fall* of vertices, ensures that the level of a vertex may only fall or remain same as the edges are inserted. Consider insertion of a cross edge (x, y) as shown in Figure 1. In order to ensure monotonic fall, the following strategy is used. If $\text{LEVEL}(y) \leq \text{LEVEL}(x)$, then we reroot $T(v)$ at y and hang it through edge (x, y) (and vice versa if $\text{LEVEL}(y) > \text{LEVEL}(x)$). This strategy surely leads to fall of level of x (or y). However, this rerooting has to be followed by transformation of $T(v)$ into a DFS tree. An obvious, but inefficient way, to do this transformation is to perform a fresh DFS traversal on $T(v)$ from x (as done by Franciosa et

al. [4]). We are able to avoid this costly step using our second principle called *minimal restructuring*. Following this principle, only a path of the subtree $T(v)$ is reversed and as a result major portion of the original DFS tree remains intact. Furthermore, this principle also facilitates monotonic fall of all vertices of $T(v)$. The rerooting procedure based on this principle is described and analyzed in the following section.

Our algorithm updates DFS tree upon insertion of any cross edge as follows. Firstly, we carry out rerooting based on the two principles mentioned above. As a result, many back edges now potentially become cross edges. All these edges are collected and virtually (re)-inserted back to the graph, and processed as *fresh* insertions. This simple iterative algorithm, when analyzed in a straightforward manner, has a time complexity $O(mn)$. However, using a more careful analysis, it can be shown that its time complexity is $O(n^{3/2}m^{1/2})$, which is strictly sub-cubic. In order to improve the time complexity further, we process the pool of cross edges in a more structured manner. In particular, we process the highest cross edge first. This leads to our final algorithm that achieves $O(n^2)$ time complexity for any arbitrary sequence of edge insertions.

4 Rerooting a Subtree

Consider insertion of an edge (x, y) which happens to be a cross edge with respect to the DFS tree T. Let w be LCA of x and y, and let u and v be the two children of w such that $x \in T(u)$ and $y \in T(v)$. See Figure 2 for a visual description. As discussed before, updating the DFS tree upon insertion of the cross edge (x, y) entails rerooting of subtree $T(v)$ at y and hanging it from x. We now describe an efficient rerooting procedure for this task based on *minimal restructuring* principle.

The underlying idea of *minimal restructuring* principle is to preserve the current tree structure as much as possible. Consider the path $P(y, v) = \langle z_1(= y), z_2, \ldots, z_k(= v)\rangle$. This path appears from v to y in the rooted tree T. Rerooting reverses this path in T so that it starts at y and terminates at v. In order to see how this reversal affects the DFS structure, let us carefully examine $T(v)$.

The subtree $T(v)$ can be visualized as a collection of disjoint trees joined through the path $P(v, y)$ as follows. Let T_1 denote the subtree $T(y)$ and let T_2 denote the subtree $T(z_2)\backslash T(z_1)$. In general, T_i denote the subtree $T(z_i)\backslash T(z_{i-1})$. Upon reversing the path $P(v, y)$, notice that each subtree remains intact but their ordering gets reversed. Furthermore, level of each subtree T_i surely falls. (see Figure 2). Let us find the consequence of reversing $P(v, y)$ on all those back edges whose at least one endpoint belongs to $T(v)$. Observe that the back edges which originate as well as terminate within the same T_i continue to remain as back edges since tree T_i remains intact. Likewise, any back edge from these subtrees which terminates at any ancestor of v also continue to remain as a back edge. However, the back edges originating in $T(v)$ and terminating on w, which were earlier stored in $\mathcal{B}(v)$, will now have to be stored in $\mathcal{B}(u)$ (recall the definition of $\mathcal{B}$). Also notice that the tree edge (w, v) now becomes a back

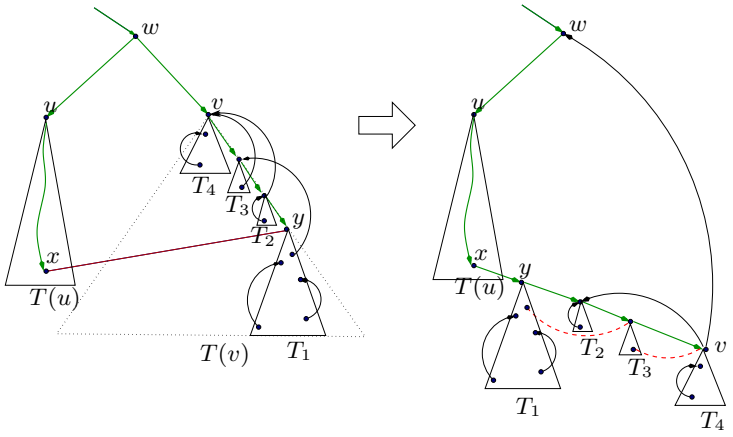

Fig. 2. Rerooting the tree $T(v)$ at y and hanging it from x. Notice that some back edges may become cross edges (shown dotted) due to this rerooting.

edge and has to be added to $\mathcal{B}(u)$. The remaining back edges are only those which originate from some T_i and terminate at some $z_j, j > i$. All these edges are present in $\mathcal{B}(z_{j-1})$. Some of these back edges may become cross edges due to the reversal of $P(v, y)$. Their presence violates the DFS property (relation $\mathcal{R}$) of the new tree. We just collect and remove these edges temporarily from the graph. In summary, our rerooting algorithm just does the following: It traverses the path $P(v, y)$ from y to v, collects $\mathcal{B}(z_j)$ for each $1 \leq j < k$, and reverses the path $P(v, y)$. The pseudo code of the rerooting process is described in Procedure `Reroot`.

Procedure Reroot(v,x,y): reroots subtree $T(v)$ at vertex y and hangs it through edge (x, y).

```
1  E_R ← φ;
2  z ← y;
3  p ← x;
4  B(v) ← B(u) ∪ B(v);
5  while z ≠ par(v) do
6  |   if z ≠ v then E_R ← E_R ∪ B(z);
7  |   B(z) ← φ;
8  |   next ← par(z);
9  |   par(z) ← p;
10 |   p ← z;
11 |   z ← next;
12 end
13 Return E_R
```

The following lemma holds based on the above discussion.

Lemma 1. *Tree T at the end of Procedure* `Reroot`(v, x, y) *is a DFS tree for the graph* $(V, E \backslash \mathcal{E}_R)$.

We introduce some terminology to facilitate compact and clean reference to the entities of rerooting procedure. The lower and higher end vertices x and y of the inserted edge (x, y) are called *prime* and *conjugate* vertices respectively. Notice that the restructured subtree now hangs from the *prime* vertex. We define *prime* path as the path from the prime vertex x to u and *conjugate* path as the path from conjugate vertex y to v, where u and v are children of $LCA(x, y)$ s.t. $x \in T(u)$ and $y \in T(v)$.

Each vertex of subtree $T(v)$ suffers a fall in its level due to `Reroot`(v, x, y). We shall now calculate this fall exactly. Let $\Delta = \text{LEVEL}(x) - \text{LEVEL}(y)$. As a result of the rerooting, y has become child of x. Hence it has suffered a fall in its level by $\Delta + 1$. Since $T_1 = T(y)$ and T_1 remains intact, so each vertex of T_1 suffers a fall by $\Delta + 1$ levels. Consider a vertex z_i which is the root of T_i for some $i > 1$. This vertex was earlier at level $i - 1$ higher than $y (= z_1)$ and now lies at $i - 1$ level below y. Hence overall level of z_i (and hence that of every vertex of T_i) has fallen by $\Delta + 2i - 1$. This leads us to the following lemma.

Lemma 2. *Let Δ be the difference in the levels of prime and conjugate vertices before rerooting. After rerooting, the ith vertex on the conjugate path falls by $\Delta + 2i - 1$ levels.*

Let us analyze the time complexity of the Procedure `Reroot`(v, x, y). It first adds $\mathcal{B}(v)$ to $\mathcal{B}(u)$; this step takes $O(1)$ time since we are uniting two lists. Thereafter, the procedure traverses the conjugate path $P(y, v)$, and collects the edges $\mathcal{B}(z)$ for each $z \in P(y, v) \backslash \{v\}$ in $\mathcal{E}_R$.

Lemma 3. *The time complexity of the Procedure* `Reroot`(v, x, y) *is $O(k + |\mathcal{E}_R|)$, where k is the length of the conjugate path and $\mathcal{E}_R$ is the set of edges returned by the procedure.*

It follows from the rerooting procedure that any back edge getting converted to a cross edge is surely collected in $\mathcal{E}_R$. However, not all the edges of $\mathcal{E}_R$ necessarily become cross edges. In order to understand this subtle point, observe that $\mathcal{E}_R$ contains all those edges which originate from some vertex in T_i and terminate at some $z_j, i < j < k$. Consider any such edge (a, z_j), $a \in T_i$. If $a \neq z_i$ (root of T_i), then surely (a, z_j) has become a cross edge after reversal of $P(v, y)$. But if $a = z_i$, then it still remains a back edge. So we can state the following lemma which will be crucial in our final algorithm.

Lemma 4. *If an edge collected in $\mathcal{E}_R$ is a back edge with respect the modified DFS tree, then both its endpoints must belong to the conjugate path.*

5 Algorithm for Incremental DFS

Consider insertion of an edge (t, z). In order to update the DFS tree, our algorithm maintains a set $\mathcal{E}$ of edges which is initialized as $\{(t, z)\}$. The algorithm

then processes the set $\mathcal{E}$ iteratively as follows. In each iteration, an edge (say (x, y)) is extracted from $\mathcal{E}$ using Procedure `Choose`. If the edge is a back edge, the edge is inserted in the set of back edges $\mathcal{B}$ accordingly and no processing is required. If (x, y) is a cross edge, it is processed as follows. Let w be LCA of x and y, and let v be the child of w such that y is present in subtree $T(v)$. Without loss of generality, let $\text{LEVEL}(x) \geq \text{LEVEL}(y)$. Procedure `Reroot`$(v, x, y)$ is invoked which reroots subtree $T(v)$ at y and returns a set of edges collected during the procedure. All these edges are extracted from E and added to $\mathcal{E}$. This completes one iteration of the algorithm. The algorithm finishes when $\mathcal{E}$ becomes empty. The correctness of the algorithm follows directly from the following invariant which is maintained throughout the algorithm:

Invariant: T is DFS tree for the subgraph $(V, E \backslash \mathcal{E})$.

Algorithm 1: Processing insertion of an edge (t, z)

```
1 E ← {(t, z)} ;            /* E is a set of edges to be inserted. */
2 while E ≠ φ do
3 |  (x, y) ← Choose(E) ;          /* Here LEVEL(x) ≥ LEVEL(y). */
4 |  w ← LCA(x, y);
5 |  v ← LA(y, LEVEL(w) + 1) ; /* v is y's ancestor & w's child */
6 |  if w ≠ y then                  /* (x, y) is a cross edge. */
7 |  |  E ← E ∪ Reroot(v, x, y);
8 |  end
9 end
```

Procedure Choose($\mathcal{E}$): Chooses and returns an edge from $\mathcal{E}$.

Remove an arbitrary edge (x, y) from $\mathcal{E}$.
Return (x, y).

5.1 Analysis

The computation cost of collecting and processing each edge $e \in \mathcal{E}$ can be associated with the rerooting event in which it was collected. Thus the computation time spent by the incremental algorithm in processing any sequence of edge insertions is of the order of the time spent in all the rerooting calls invoked. Furthermore, using Lemma 3 we know that the time complexity of a single rerooting call is of the order of the number of edges in $\mathcal{E}_R$ that were collected during that rerooting (the cost of the first term mentioned in Lemma 3 can be associated with the fall of vertices on the conjugate path). Therefore, in order to calculate the time complexity of the algorithm, it suffices to count all the edges collected during various rerooting calls. Note that this count can be much larger than $O(m)$ because an edge can appear multiple times in $\mathcal{E}$ during the algorithm. It follows from Lemma 2 that whenever an edge is collected during

a rerooting call, the level of at least one of its endpoints falls. Since level can fall only up to n, it follows that the computation associated with a single edge during the algorithm is of the order of n. Hence, the $O(mn)$ time complexity of the algorithm is immediate. However, using a more careful insight into the rerooting procedure, we shall now show that the time complexity is much better.

Consider any execution of the rerooting procedure. Let $((y =)z_1, z_2, \ldots, z_k(= v))$ be the path that gets reversed during the rerooting process. See Figure 2. The procedure collects the edges $\mathcal{B}(z_i)$ for each $i \leq k$. We shall now *charge* each edge collected to the fall of one of its endpoints. Let τ be a parameter whose value will be fixed later. Consider any edge (a, z_i) that is collected during the rerooting process. Note that level of each of a and z_i has fallen due to the rerooting. If $i \leq \tau$, we charge this edge to the fall of vertex a. In this way, there will be at most τ edges that get charged to the fall of a. If $i > \tau$, we charge this edge to the fall of z_i. It follows from Lemma 2 that z_i falls by at least $2i - 1 > \tau$ levels in this case.

Consider any vertex s in the graph. Over any sequence of edge insertions, if s falls by less than τ levels, we call it a *small* fall; Otherwise we call it a *big* fall for s. It follows that s will be charged τ edges for every small fall. The number of small falls is $O(n)$, so overall cost charged to s due to all its small falls is $O(n\tau)$. On the other hand, s will be charged $O(\deg(s))$ for every big fall. Notice that there will be at most n/τ big falls of s throughout any sequence of edge insertions. So the overall cost charged to all big falls of s will be $O(\deg(s) \cdot n/\tau)$. Hence for all vertices, the total computation charged will be $O(n^2\tau + mn/\tau)$. Fixing $\tau = \sqrt{m/n}$, we can conclude that the overall computation performed during processing of any sequence of m edge insertions by the algorithm is $O(n^{3/2}m^{1/2})$.

Theorem 3. *For an undirected graph G on n vertices, a DFS Tree can be maintained under insertion of any arbitrary sequence of edges with total update time of $O(n^{3/2}\sqrt{m})$.*

Note: Our algorithm requires us to answer LCA and LA queries efficiently. Data structure for LCA and LA maintained for T are subject to insertion and deletion of edges in T. These can be simulated using $O(n^2)$ leaf updates taking $O(n^2)$ time throughout the algorithm. For details, refer to appendix.

It follows from Theorem 3 that even for any sequence of $\Theta(n^2)$ edge insertions, the total update time in maintaining DFS tree is $O(n^{2.5})$ which is strictly subcubic. In fact, with a more structured way of processing the edges of $\mathcal{E}$, we can even achieve a bound of $O(n^2)$. We provide this improved algorithm in the following section.

6 Achieving $O(n^2)$ update time

The time complexity of Algorithm 1 is governed by the number of edges in $\mathcal{E}$ that are processed during the algorithm. In order to get an improved algorithm, let us examine $\mathcal{E}$ carefully. An edge from $\mathcal{E}$ can be a cross edge or a back edge. Processing of a cross edge always triggers a rerooting event which, in turn, leads

to fall of one or more vertices. Hence, the total number of cross edges processed during the algorithm is $O(n^2)$. All the remaining edges processed in $\mathcal{E}$ during the entire algorithm are back edges. There are two sources of these back edges.

Firstly, some edges added to $\mathcal{E}$ by Procedure Reroot are back edges. Let us analyze their count throughout the algorithm. It follows from Lemma 4 that both endpoints of each such back edge belong to the conjugate path associated with the rerooting call. Notice that ith vertex on the conjugate path falls by at least $2i-1$ levels (see Lemma 2). So, if ℓ is the length of the conjugate path, the total fall in the level of all vertices on the conjugate path is more than $\ell(\ell-1)/2$ which is an upper bound on the number of edges with both endpoints on the conjugate path. Since the total fall in the level of vertices cannot be more than $O(n^2)$, the number of such back edges throughout the algorithm is $O(n^2)$.

Secondly, some edges added to $\mathcal{E}$ by Procedure Reroot are cross edges at the time of their collection, but become back edges before they are processed. This may happen due to rerooting initiated by some other cross edge from $\mathcal{E}$. In order to understand this subtle point, see Figure 3. Here $e_1, e_2, e_3, e_4, e_5 = (x, y)$ are cross edges present in $\mathcal{E}$ at some stage. While we process (x, y), $T(v)$ gets rerooted at y and hangs through edge (x, y). As a result e_1, e_2, e_3, e_4 become back edges.

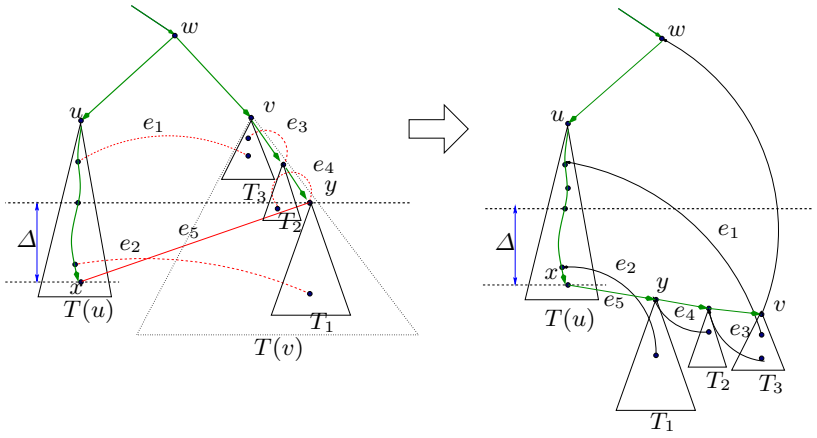

Fig. 3. Some cross edges in $\mathcal{E}$ become back edges due to the rerooting of $T(v)$.

In order to bound these cross edges getting transformed into back edges during rerooting of $T(v)$, let us carefully examine one such cross edge. Let v_h and v_l be respectively the higher and lower endpoints of the resulting back edge. The edge (v_h, v_l) was earlier a cross edge, so v_h now has a new descendant v_l. Similarly v_l now has a new ancestor v_h. Note that the descendants of only vertices lying on prime and conjugate paths are changed during rerooting. Also the ancestors of only the vertices in $T(v)$ are changed during rerooting. Hence the following lemma holds true.

Lemma 5. *Cross edges getting converted to back edges as a result of rerooting $T(v)$ are at most $|T(v)|$ times the sum of lengths of prime and conjugate paths.*

Observe that the total sum of fall of vertices of $T(v)$ during the rerooting event is at least $|T(v)| \cdot (\Delta + 1)$ (see Figure 3). However, the sum of lengths of prime and conjugate paths may be much greater than $\Delta+1$. Hence, the number of cross edges getting transformed into back edges is not bounded by the fall of vertices of $T(v)$. This observation, though discouraging, also gives rise to the following insight: If there were no cross edges with level higher than LEVEL(y), the number of cross edges converted to back edges will be fewer. This is because the possible higher endpoints of such edges on the prime or conjugate path will be limited. This insight suggests that processing higher cross edges from $\mathcal{E}$ first will be more advantageous than lower ones. Our final algorithm is inspired by this idea.

6.1 The final algorithm

Our final algorithm is identical to the first algorithm except that instead of invoking Procedure `Choose` we invoke Procedure `ChooseHigh`. It processes the edges of set $\mathcal{E}$ in non-increasing order of their levels. To extract the highest edge from $\mathcal{E}$, we may use a binary heap on endpoints of these edges taking $O(\log n)$ time per operation. However, this can be improved to $O(1)$ amortized time using a much simpler data structure that exploits the following facts. Firstly, level of a vertex is an integer in $[1, n]$. Secondly, when a rerooting event occurs upon insertion of an edge e, all the edges collected are at a level lower than LEVEL(e). Kindly refer to appendix for details of this data structure.

Procedure ChooseHigh($\mathcal{E}$): Chooses and returns the highest edge from $\mathcal{E}$.

Remove the highest edge (x, y) from $\mathcal{E}$.
Return (x, y).

6.2 Analysis

In order to establish $O(n^2)$ bound on the running time of our final algorithm, it follows from the preceding discussion that we just need to show that the number of cross edges getting converted to back edges throughout the algorithm is $O(n^2)$.

Consider any rerooting event initiated by cross edge (x, y) (refer to Figure 3). Since there is no edge in $\mathcal{E}$ which is at a higher level than LEVEL(y), so it follows from Lemma 5 that the cross edges getting converted to back edges during the rerooting event will be of one of the following types only.

- The cross edges with one endpoint in $T(v)$ and another endpoint x or any of Δ ancestors of x.
- The cross edges with y as one endpoint and another endpoint anywhere in $T(v)\backslash T(y)$.

Hence the following lemma holds for our final algorithm.

Lemma 6. *During the rerooting event the number of cross edges converted to back edges are at most* $|T(v)| \cdot (\Delta + 1) + |T(v) \backslash T(y)|$.

According to Lemma 2, level of each vertex of $T(v)$ falls by at least $\Delta + 1$. So the first term in Lemma 6 can be clearly associated with the fall of each vertex of $T(v)$. Note that each vertex in $T(v) \backslash T(y)$ becomes a descendant of y and hence falls by at least one extra level (in addition to $\Delta + 1$). This fall by extra one or more levels for vertices of $T(v) \backslash T(y)$ can be associated with the second term mentioned in Lemma 6. Hence the total number of cross edges getting transformed to back edges during the algorithm is of the order of $O(n^2)$. We can thus conclude with the following theorem.

Theorem 4. *For an undirected graph G on n vertices, a DFS Tree can be maintained under insertion of any arbitrary sequence of edges with total update time of $O(n^2)$.*

7 Conclusions

We presented a simple algorithm for maintaining DFS tree under insertion of edges. The $O(n^2)$ bound of our algorithm is quite tight. In fact, we can show the following: There exists a sequence of $\Theta(n)$ edge insertions such that every incremental algorithm for maintaining DFS tree that follows the principle of *monotonic fall* will require $\Omega(n^2)$ time.

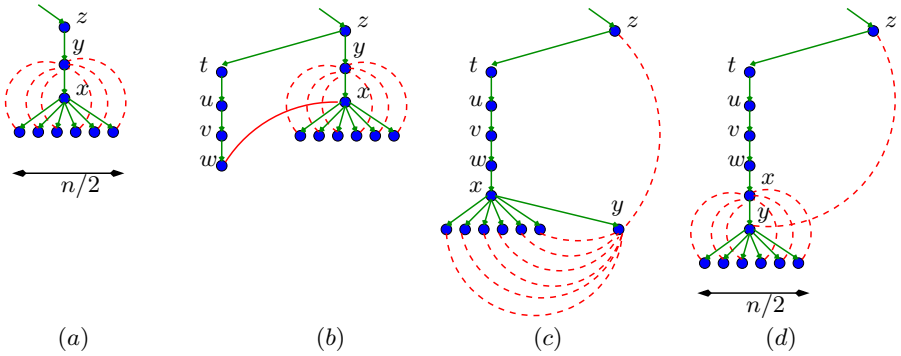

Fig. 4. Example to show tightness of analysis.

Consider a graph on n vertices with $n/2-3$ singleton vertices in the beginning, and a set S of $n/2$ vertices hanging from a vertex x. Let $par(x) = y$ and $par(y) = z$, and each vertex in S has a back edge to y (see Figure 4(a)). Now use 4 singleton vertices t, u, v, w, and add 5 edges $(z, t), (t, u), (u, v), (v, w), (w, x)$ in this order (Figure 4(b)). If the algorithm follows *monotonic fall*, it must hang y from x (Figure 4(c)) and then eventually hang S from y (Figure 4(d)). This creates a

structure similar to the original Figure 4(a). This process changes $n/2$ tree edges and hence requires atleast $\Omega(n)$ time. This step can be repeated $n/8$ times. Hence overall inserting $\Theta(n)$ edges will take $\Omega(n^2)$ time.

It would be interesting to see whether DFS tree can be maintained incrementally in $o(n^2)$ time for sparse graphs. It follows from the above example that a new approach might be required to achieve this goal.

References

1. Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *ICALP*, pages 73–84, 2000.
2. Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.
3. Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981.
4. 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.
5. Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994.
6. John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.
7. John H. Reif. A topological approach to dynamic graph connectivity. *Inf. Process. Lett.*, 25(1):65–70, 1987.
8. Liam Roditty and Uri Zwick. On dynamic shortest paths problems. In *ESA*, pages 580–591, 2004.
9. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

Appendix

Maintaining LCA and LA dynamically

Algorithm 1 requires us to answer LCA and LA queries efficiently. The structures introduced in Theorem 1 and Theorem 2 maintain LCA and LA in the dynamic setting. These structures allow only leaf updates in the underlying tree (in our case T). However, Procedure `Reroot` (see Figure 2) inserts an edge (x, y) and deletes an edge (w, v) in T that leads to rerooting the subtree $T(v)$ at the vertex y. These edges may not be the leaf edges hence these operations are not directly supported by the structure.

To perform these updates first iteratively leaves of $T(v)$ are deleted until whole of $T(v)$ is removed. Now the subtree $T(y)$ is rebuilt at y iteratively by a series of leaf insertions. We know that each vertex in $T(v)$ falls by at least one level during the rerooting event. Note that each falling vertex leads to exactly one leaf insertion and deletion during this update process each taking $O(1)$ amortized time. Since each vertex can fall at most n times, the overall time taken to maintain this data structure (for maintaining LCA and LA) throughout the algorithm is $O(n^2)$.

Data Structure to maintain set of edges $\mathcal{E}$

Our algorithm requires the edges in $\mathcal{E}$ to be processed in non-decreasing order of their levels. So we require a data structure that maintains $\mathcal{E}$ and allows retrieval of edges in the desired order. This data structure also supports insertion of new edges and change in the level of edges due to fall of a vertex.

A binary max-heap on edges $\mathcal{E}$ could be used here with level of edge as the key. But it may require $degree(v)$ updates when level of a vertex v is changed. In order to avoid $degree(v)$ updates, we may maintain the binary heap storing end vertices of edges in $\mathcal{E}$. This will allow each operation to be performed in $O(\log n)$ time. In this manner, we can maintain $\mathcal{E}$ in overall $O(n^2 \log n)$ time during the entire algorithm. However, it can be improved to $O(n^2)$ using a very simple data structure that exploits the following facts. Firstly, level of a vertex is an integer ($[1, n]$). Secondly, when a rerooting event occurs on insertion of an edge e, the edges collected during the event are at a level lower than $\textsc{level}(e)$.

Our data structure is an array H where $H[i](i \in [1, n])$ stores all vertices present at level i with at least one edge from $\mathcal{E}$ incident on it. Further, each vertex stores a list of edges present in $\mathcal{E}$ incident on it. Hence insertion of edges and updating the level of a vertex can be performed in $O(1)$ time.

Consider the insertion of an edge e to the graph. Insertion of this edge may lead to a rerooting event in which several edges are collected in $\mathcal{E}$ and hence added to H. In order to process these edges (or corresponding end vertices) in non-decreasing order of their level we start a scan of H from $\textsc{level}(e)$. Notice that all the edges collected in future rerooting events caused by some edge e' in $\mathcal{E}$ will be added in H at a level lower than the $\textsc{level}(e')$. Hence the next highest edge will always be found as the scan progresses. The scan is stopped when $\mathcal{E}$ is empty.

The total time required by the data structure throughout the algorithm is of the order of number of edges processed and the time required for traversing through H. The first term uses $O(n^2)$ time as analysed is Section 5. Cost of traversing through H can be easily associated with fall of the vertex suffering maximum fall during a rerooting event. Hence the overall time required to maintain the data structure is $O(n^2)$.