Algorithmica (2017) 79:466–483
DOI 10.1007/s00453-016-0204-1

Incremental Algorithm for Maintaining a DFS Tree for Undirected Graphs

Surender Baswana[1] · **Shahbaz Khan**[1]

Received: 26 December 2015 / Accepted: 16 August 2016 / Published online: 26 August 2016
© Springer Science+Business Media New York 2016

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. 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 nineteenth century French mathematician who used it for solving mazes. However, it was Tarjan, who in his seminal work [22], demonstrated the power of DFS traversal for solving vari-

A preliminary version of this result appeared in ICALP 2014 [3].
This research work was partially supported by UGC-ISF (the University Grants Commission of India and Israel Science Foundation), IMPECS (the Indo-German Max Planck Center for Computer Science), and Google India under the Google India PhD Fellowship Award.

✉ Shahbaz Khan
shahbazk@cse.iitk.ac.in
http://www.cse.iitk.ac.in/users/shahbazk/

Surender Baswana
sbaswana@cse.iitk.ac.in
http://www.cse.iitk.ac.in/users/sbaswana/

[1] Department of CSE, IIT Kanpur, Kanpur, India

ous fundamental graph problems, namely, topological sorting, connected components, biconnected components and strongly-connected components. Since then, DFS traversal is one of the most widely used graph traversal techniques. Interestingly, the role of DFS traversal is not confined to merely the design of efficient algorithms. For example, consider the classical result of Erdős and Rényi [8] for the phase transition phenomena in random graphs. There exist many proofs of this result which are intricate and based on highly sophisticated probability tools. However, recently, Krivelevich and Sudakov [14] designed a truly simple, short, and elegant proof for this result based on the insights from a DFS traversal in a graph.

A 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). For a given graph, there may exist many DFS trees rooted at a given vertex $r \in V$. 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 tree problem is to compute the order in which the vertices get visited during this restricted traversal.

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

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

This relation defined by a DFS tree has played the key role in solving various graph problems. Similar relations exist for a DFS tree in a directed graph.

Most of the graph applications in the real world deal with graphs that keep changing with time. An algorithmic graph problem is modeled in the dynamic environment as follows. There is an online sequence of insertion and deletion of edges and the aim is to maintain the solution of the given problem after every edge update. To achieve this aim, we need to maintain some clever data structure for the problem such that the time taken to update the solution after an edge update is much smaller than that of the best static algorithm. A dynamic graph algorithm is said to be fully dynamic if it handles both insertion as well as deletion updates. A partially dynamic algorithm is said to be incremental or decremental if it handles only insertion or only deletion of edges respectively. In the last two decades, many elegant dynamic algorithms have been designed for various graph problems such as connectivity [7,11–13], reachability [19,21], shortest path [6,20], spanner [4,10,18], and min-cut [23].

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 [16,17] was the first to address the complexity of the DFS problem in a dynamic environment. He showed that the ordered DFS tree problem is one of the hardest problems to dynamize in the entire class P. Later, Milterson et al. [15] introduced a class of problems called non-redundant polynomial (*NRP*) complete. They showed that if the solution of any *NRP*-complete problem is updateable in $O(\text{polylog}(n))$ time, then the solution of every problem in the class P is updateable 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 tree 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

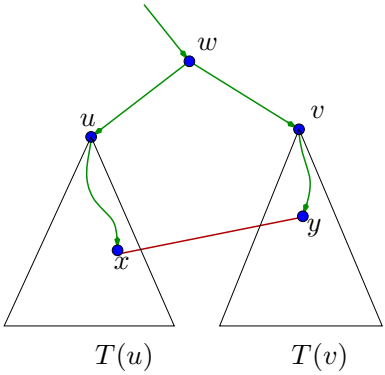

Fig. 1 (x, y) is a cross edge

any DFS tree in a dynamic environment. For the case of a directed acyclic graph (DAG), Franciosa et al. [9] presented an incremental algorithm for maintaining a DFS tree in $O(mn)$ total time. Recently, again only for a DAG, Baswana and Choudhary [2] presented a decremental algorithm for maintaining a DFS tree in expected $O(mn \log n)$ total time. These are the only non-trivial results available for the dynamic DFS tree problem. Maintaining a DFS tree incrementally for an undirected graph (or general directed graph) was stated as an open problem by Franciosa et al. [9]. 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 is required in the DFS tree. Such an edge is called a *back edge*. Otherwise, the relation $\mathcal{R}(x, y)$ does not hold for the edge (x, y), and we call such an edge a *cross* edge. See Fig. 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 $y \in T(v)$, where $T(u)$ and $T(v)$ are the subtrees of the DFS tree rooted at u and v respectively. Insertion of the edge (x, y) violates the property of a DFS tree as follows. Let S be the set of visited vertices when the DFS traversal reaches 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 $V \setminus S$. However, the 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 in the DFS tree. This implies that $T(u)$ will have to be rerooted at x and hung from y (or $T(v)$ will have 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 edge in $T(u)$. It is not obvious how to perform this restructuring in an efficient manner.

1.1 Our Result

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. In order to handle insertion of cross edges efficiently, we use two principles. The first principle, called *monotonic fall*,

ensures that the depth of each vertex never decreases during the algorithm. The second principle, *minimal restructuring*, ensures that while rerooting a subtree upon insertion of a cross edge, we perform minimal changes in the subtree in order to preserve the relation $\mathcal{R}$. While none of these two principles in isolation is effective, it is their novel combination that leads to an efficient incremental algorithm for a DFS tree. Observe that the amortized update time per edge insertion is $\Omega(n^2/m)$, which is $O(1)$ for the case $m = \Theta(n^2)$.

A standard 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. Our algorithm maintains a DFS tree explicitly at each stage. Baswana and Choudhary [2] recently established a worst case lower bound of $\Omega(mn)$ for maintaining the ordered DFS tree explicitly under insertion (or deletion) of edges. In the light of this lower bound, our algorithm implies that maintaining a DFS tree explicitly in the incremental environment is provably faster than maintaining an ordered DFS tree for dense graphs.

We also show the existence of a sequence of $\Theta(n)$ edge insertions such that any incremental algorithm that obeys the principle of *monotonic fall* must require $\Omega(n^2)$ time for maintaining a DFS tree explicitly. Therefore, the $O(n^2)$ time complexity of our algorithm is indeed tight even for sparse graphs.

Furthermore, our algorithm uses only $O(m + n)$ extra space. Excluding the standard data structures for maintaining ancestors in a rooted tree [1,5], our algorithm employs very simple data structures. These salient features make our algorithm an ideal candidate for practical applications as well.

1.2 Organization of the Article

In Sect. 2, we describe various notations and tools used throughout our paper. Section 3 presents the overview of our algorithm including the two principles used by it. Then in Sect. 4, we describe the rerooting procedure based on these principles. With this rerooting procedure at its core, we describe our incremental algorithm gradually in two steps. First we describe a basic incremental algorithm that achieves overall $O(n^{3/2}m^{1/2})$ time in Sect. 5. In Sect. 6, we describe and analyze our main algorithm. Interestingly, this algorithm is obtained by a small tweak to the basic incremental algorithm; however, its careful analysis establishes a bound of $O(n^2)$ on its time complexity. We later establish the tightness of the upper bound of our algorithm.

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 time during the algorithm.
- r : Root of the tree T.
- $par(v)$: Parent of v in T.
- $P(u, v)$: Path between u and v in T.

- LEVEL(v) : Level of a vertex v in T such that LEVEL(r) $= 0$, and LEVEL(v) $=$ LEVEL($par(v)$) $+ 1$.
- LEVEL(e) : Level of an edge $e = (x, y)$ in T such that LEVEL(e) $=$ min(LEVEL(x), 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 in the downward direction, a vertex u is said to be at higher level than vertex v if LEVEL(u) $<$ LEVEL(v). Similarly an edge e is said to be higher than edge e' if LEVEL(e) $<$ LEVEL(e').

We also maintain the following information about the DFS tree T during the algorithm.

- Each vertex v keeps a pointer to $par(v)$. For the root r, $par(r) = r$.
- For each $v \in T$, we keep a list of all its children in tree T. This facilitates the traversal of $T(v)$ from the vertex v.
- Each vertex v keeps a list $\mathcal{B}(v)$ which consists of all the 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. $\mathcal{B}(v)$ is maintained as a circular linked list to enable merging of two lists in $O(1)$ time.

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

Theorem 1 [5] *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 [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 the Level Ancestor problem, as stated in Theorem 2, can be easily extended to handle the deletion of a leaf node in amortized $O(1)$ time using the standard technique of periodic rebuilding.

If the graph is not connected, we need 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 the 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 the same as the edges

are inserted. Consider insertion of a cross edge (x, y) as shown in Fig. 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). Otherwise, we reroot $T(u)$ at x and hang it through edge (y, x). This strategy surely leads to a fall in the 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. [9] in case of DAG. 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. In fact, 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 in a pool, virtually (re)-inserted back into the graph one by one, 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. We also present a worst case example proving the tightness of this analysis. 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. Subsequently, we also prove that this is the best possible time complexity even for sparse graphs that can be achieved by any algorithm that is based on *monotonic fall* . To establish this fact, we present a sequence of $\Theta(n)$ edge insertions such that any such algorithm would require $\Omega(n^2)$ total time.

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 $y \in T(v)$. Let $\text{LEVEL}(y) \leq \text{LEVEL}(x)$. See Fig. 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 the principle of *minimal restructuring*.

The underlying idea of *minimal restructuring* 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 hanging from 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 denotes the subtree $T(z_i)\backslash T(z_{i-1})$. Upon reversing the

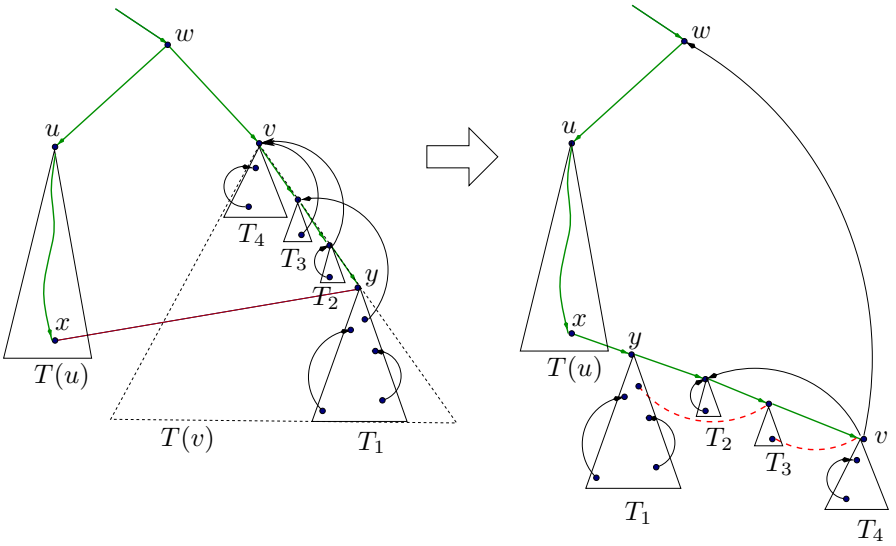

Fig. 2 Rerooting the tree $T(v)$ at y and hanging it from x. Notice that some back edges may become cross edges (shown *dotted*) due to this rerooting

path $P(v, y)$, notice that each subtree remains intact but their ordering gets reversed. Furthermore, the level of each vertex in every subtree T_i surely falls. (see Fig. 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 continues to remain a back edge. However, the back edges originating in $T(v)$ and terminating on w (i.e., $LCA(x, y)$), which were earlier stored in $\mathcal{B}(v)$, will now have to be stored in $\mathcal{B}(u)$. Recall that $\mathcal{B}(v)$ contains the back edges that originate from $T(v)$ and terminate at $par(v)$. Also, notice that the tree edge (w, v) now becomes a back 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 $P(v, y)$ (see Fig. 2). Their presence violates the DFS property (relation $\mathcal{R}$) of the new tree. We simply collect these edges in a set $\mathcal{E}_R$ and remove them 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 $1 \leq j < k$ in $\mathcal{E}_R$, and reverses the path $P(v, y)$. The pseudocode of the rerooting process is described in Procedure Reroot.

The following lemma holds based on the above discussion.

Lemma 1 *Tree T at the end of Procedure Reroot 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 the rerooting procedure. The lower and higher end vertices x and y of the

Procedure Reroot(u,v,x,y): reroots subtree $T(v)$ at vertex y and hangs it through edge (x, y). It also updates the data structure $\mathcal{B}$ for every vertex on $P(y, v)$ and u, where u is the sibling of v s.t. $x \in T(u)$. It returns the set of back edges $\mathcal{E}_R$ which could potentially be cross edges after rerooting.

```
1  B(u) ← B(u) ∪ B(v) ∪ {(par(v), v)};
2  E_R ← φ;
3  z ← y ;                          /* z is the first vertex of path P(y, v) */
4  p ← x;
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 ;                          /* updating the parent of z */
10 |  p ← z;
11 |  z ← next ;                 /* z is now the next vertex on path P(y, v) */
12 end
13 Return E_R
```

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(u, 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 also suffers the same fall (of $\Delta + 1$ levels) as y. 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 (starting from the conjugate vertex) falls by $\Delta + 2i - 1$ levels.*

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

Lemma 3 *The time complexity of the Procedure Reroot 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 collected in $\mathcal{E}_R$ have necessarily become cross edges after Procedure Reroot. 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 to the modified DFS tree, then both its endpoints must belong to the conjugate path.*

5 Algorithm for Incremental DFS

We now describe our algorithm for incremental maintenance of a DFS tree. Consider the 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$(u, v, x, y)$ 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 terminates when $\mathcal{E}$ becomes empty (see Algorithm 1 for pseudocode). 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) ;                          /* Let level(x) ≥ level(y). */
4  |  w ← LCA(x, y);
5  |  u ← LA(x, LEVEL(w) + 1) ;          /* u is the child of w s.t. x ∈ T(u) */
6  |  v ← LA(y, LEVEL(w) + 1) ;          /* v is the child of w s.t. y ∈ T(v) */
7  |  if w ≠ y then                                  /* (x, y) is a cross edge. */
8  |  |  E ← E ∪ Reroot(u, v, x, y);
9  |  end
10 end
```

Furthermore, the LCA and LA data structures introduced in Theorems 1 and 2 have to be updated after every iteration of the algorithm. Also, since we maintain the level of each vertex explicitly, $\text{LEVEL}(z)$ has to be updated for each $z \in T(v)$. Following section describes how to perform these updates efficiently.

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 Maintaining LCA and LA Dynamically

Algorithm 1 requires us to answer LCA and LA queries efficiently. The data structures introduced in Theorems 1 and 2 maintain LCA and LA in the dynamic setting and answer each query in $O(1)$ time. These data structures allow only leaf updates in the underlying tree (in our case T). However, Procedure Reroot 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 (see Fig. 2). These edges may not be the leaf edges hence these operations are not directly supported by these data structures.

To perform these updates all the edges in $T(v)$ are deleted by iteratively deleting the leaves of $T(v)$. 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 exactly one leaf deletion during this update process, each taking $O(1)$ amortized time. Also, for every falling vertex $z \in T(v)$ we can update LEVEL(z) in $O(1)$ time. Since each vertex can fall at most n times during the algorithm (ensured by *monotonic fall*), the overall time taken to maintain these data structures (for maintaining LEVEL, LCA and LA) throughout the algorithm is $O(n^2)$.

5.2 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 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 and the time spent in maintaining the data structures LEVEL, LCA and LA. Furthermore, using Lemma 3 we know that the time spent in Procedure Reroot is of the order of the number of edges in $\mathcal{E}_R$ that were collected during the rerooting event and the length of the corresponding conjugate path. However, the cost of traversing the conjugate path 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 and the cost associated with the fall of vertices. Note that this count of collected edges can be much larger than $O(m)$ because an edge can appear multiple times in $\mathcal{E}$ during the algorithm. As described earlier in Sect. 5.1, the overall cost associated with the fall of vertices is $O(n^2)$. 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 of any vertex 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 $\langle (y =) z_1, z_2, \ldots, z_k (= v) \rangle$ be the path that gets reversed during the rerooting process (see Fig. 2). The procedure collects the edges $\mathcal{B}(z_i)$ for each $i < k$. 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 v in the graph. During a single rerooting event if v falls by less than τ levels, we call it a *short* fall; otherwise we call it a *long* fall for v. It follows that v can be charged for $O(\tau)$ edges in each of its short falls. The number of short falls for v is $O(n)$, so overall cost charged to v due to all its short falls is $O(n\tau)$. On the other hand, v can be charged for $O(\deg(v))$ edges in each of its long falls. The number of long falls of v during the entire algorithm is less than n/τ. So the overall cost charged to all the long falls of v will be $O(\deg(v) \cdot n/\tau)$. 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 incrementally in $O(n^{3/2}m^{1/2})$ total time for any arbitrary sequence of edge insertions.*

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 sub-cubic. In the following section we describe an example to demonstrate the tightness of our analysis.

5.3 Tightness of Analysis

We now describe a sequence of $\Theta(m)$ edge insertions for which Algorithm 1 takes $\Theta(n^{3/2}m^{1/2})$ time. Consider a graph $G = (V, E)$ where the set of vertices V is divided into two sets V' and I, each of size $\Theta(n)$. The vertices in V' are connected in the form of a chain (see Fig. 3a) and the vertices in I are isolated vertices. Thus, it is sufficient to describe only the maintenance of DFS tree for the vertices in set V', as the vertices in I will exist as isolated vertices connected to the dummy vertex s in the DFS tree (recall that s is the root of the DFS tree).

We divide the sequence of edge insertions into n_p *phases*, where each phase is further divided into n_s *stages*. At the beginning of each *phase*, we identify three vertex sets from the set V', namely $A = \{a_1, \ldots, a_k\}$, $B = \{b_1, \ldots, b_l\}$ and $X = \{x_1, \ldots, x_p\}$, where $k, l, p \leq n$ are integers whose values will be fixed later. The value of l is p in the first phase and decreases by k in each subsequent phase. Figure 3a shows the DFS tree of the initial graph. We add pk edges of the set $A \times X$ to the graph. Clearly, the DFS tree does not change since all the inserted edges are back edges. Further, we extract two sets of vertices $C = \{c_1, \ldots, c_k\}$ and $D = \{d_1, \ldots, d_k\}$ from I and connect them in the form of a chain as shown in Fig. 3b.

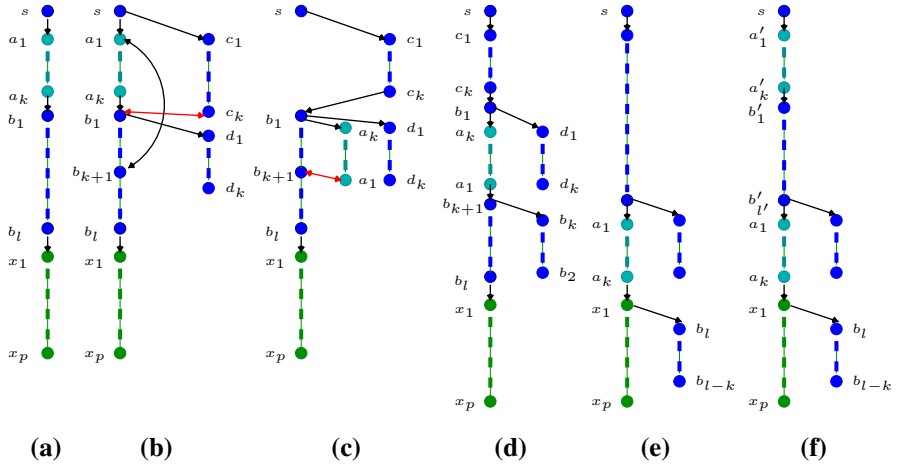

Fig. 3 Example to demonstrate the tightness of the Algorithm 1. **a** Beginning of a phase with vertex sets A, B and X. **b** Phase begins with addition of two vertex sets C and D. The first stage begins with the addition of the back edge (a_1, b_{k+1}) and the cross edge (b_1, c_k). **c** The rerooted subtree with the edges in $A \times X$ and (b_{k+1}, a_1) as cross edges. **d** Final DFS tree after the first stage. **e** Final DFS tree after first phase. **f** New vertex sets A', B' and X for the next phase

Now, the *first stage* of each phase starts with the addition of the back edge (a_1, b_{k+1}) followed by the cross edge (b_1, c_k). As a result Algorithm 1 will reroot the tree $T(a_1)$ as shown in the Fig. 3c [extra $O(1)$ vertices from I are added to C to ensure the fall of $T(a_1)$ instead of $T(c_1)$]. This rerooting makes (b_{k+1}, a_1) a cross edge. Moreover, all pk edges of set $A \times X$ also become cross edges. Algorithm 1 will collect all these edges and add them to $\mathcal{E}$ in $\Omega(pk)$ time. Since the algorithm can process the edges in $\mathcal{E}$ arbitrarily, it can choose to process (b_{k+1}, a_1) first. It will result in the final DFS tree as shown in Fig. 3d, converting all the cross edges in $A \times X$ to back edges and bringing an end to the first stage. Note the similarity between Fig. 3b, d: the set C is replaced by the set D, and the set D is replaced by the top k vertices of B. Hence, in the next stage the same rerooting event can be repeated by adding the edges (a_k, b_{2k+1}) and (b_{k+1}, d_k), and so on for the subsequent stages. Now, in every stage the length of B decreases by k vertices. Hence, the stage can be repeated $n_s = O(l/k)$ times in the phase, till A reaches next to X as shown in Fig. 3e. This completes the first phase. Now, in the next phase, first k vertices of the new tree forms the set A' followed by the vertices of B' leading up to the previous A as shown in Fig. 3f. Since the initial size of I is $\Theta(n)$ and the initial size of B is $l < n$, this process can continue for $n_p = O(l/k)$ phases. Hence, each phase reduces the size of I as well as B by k vertices.

Hence, at the beginning of each *phase*, we extract $2k$ isolated vertices from I and add pk edges to the graph. In each *stage*, we extract $O(1)$ vertices from I and add just 2 edges to the graph in such a way that will force our algorithm to process pk edges to update the DFS tree. Thus, the total number of edges added to the graph is $pk \cdot n_p$ and the total time taken by our algorithm is $pk \cdot n_p \cdot n_s$, where $n_p = O(l/k)$ and $n_s = O(l/k)$. Substituting $l = n$, $p = m/n$ and $k = \sqrt{m/n}$ we get the following theorem for any $n \leq m \leq \binom{n}{2}$.

Theorem 4 *For each value of $n \leq m \leq \binom{n}{2}$, there exists a sequence of m edge insertions for which Algorithm 1 requires $\Theta(n^{3/2}m^{1/2})$ time to maintain the DFS tree.*

Remark The core of this example is the rerooting event occurring in each stage that takes $\Theta(m^{3/2}/n^{3/2})$ time. This event is repeated systematically $n_s \cdot n_p$ times to force the algorithm to take $\Theta(n^{3/2}m^{1/2})$ time. However, this is possible only because our algorithm processes $\mathcal{E}$ arbitrarily: processing the cross edge (b_k, a_1) first amongst all the collected cross edges. Note that, had the algorithm processed any other cross edge first, we would have reached the end of a phase in a single stage. The overall time taken by the algorithm for this example would then be just $\Theta(m)$. Interestingly, with just a more structured way of processing the edges of $\mathcal{E}$, we can even achieve a worst case bound of $O(n^2)$ for our algorithm. 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 event. 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 Fig. 4. Here e_1, e_2, e_3, e_4 and $e_5 = (x, y)$ are cross edges present in $\mathcal{E}$ at some stage. While we process (x, y), $T(v)$ gets rerooted at y and hangs through the edge (x, y). As a result e_1, e_2, e_3 and e_4 become back edges.

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.

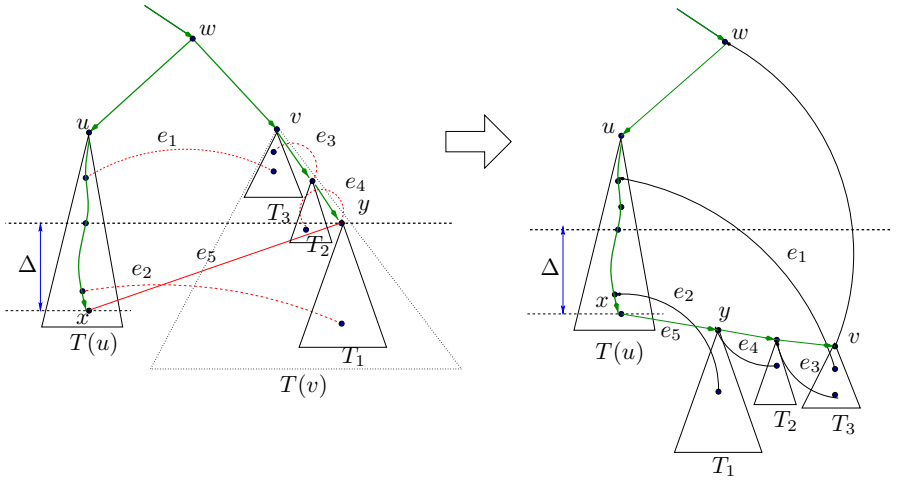

Fig. 4 Some cross edges in $\mathcal{E}$ become back edges due to the rerooting of $T(v)$

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 the fall of vertices of $T(v)$ during the rerooting event is at least $|T(v)| \cdot (\Delta + 1)$ (see Fig. 4). However, the sum of the 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 total sum of 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 at most the total sum of fall of vertices of $T(v)$. 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 the lower ones. Our final algorithm is inspired by this idea.

6.1 The Final Algorithm

Our final algorithm is identical to the previous algorithm in Sect. 5 except that instead of invoking Procedure Choose we invoke Procedure ChooseHigh.

Procedure ChooseHigh($\mathcal{E}$): Chooses and returns the highest edge in $\mathcal{E}$.
Remove the highest edge (x, y) from $\mathcal{E}$. Return (x, y).

The algorithm thus processes the edges of set $\mathcal{E}$ in a non-decreasing order of their levels. To achieve this we require a data structure that maintains $\mathcal{E}$ and allows retrieval of edges in the desired order. This data structure should also support insertion of new edges and change in the level of edges due to fall of a vertex efficiently.

This can be achieved by keeping a binary heap on endpoints of the edges in $\mathcal{E}$, where the *key* of each endpoint is its level in T. Thus, each of the two operations mentioned above can be performed in $O(\log n)$ time taking total $O(n^2 \log n)$ time throughout the algorithm. However, this can be improved to total $O(n^2)$ time using a simple data structure that is described in the following section.

6.2 Data Structure to Maintain Set of Edges $\mathcal{E}$

Our data structure is an array H where $H[i]$, $i \in [1, n]$, stores a list of all the vertices at level i that have at least one of its edges in $\mathcal{E}$. In addition, each vertex v stores a list $\mathcal{L}(v)$ of all its edges that are present in $\mathcal{E}$. It can be observed that upon insertion of an edge in $\mathcal{E}$ or update in the level of a vertex in T, H can be updated in $O(1)$ time.

Consider insertion of any cross edge, say $e = (x, y)$ in the graph. Initially the data structure H is empty. We insert x and y at their corresponding levels in H. We then add e to both $\mathcal{L}(x)$ and $\mathcal{L}(y)$. We now start a scan of H from LEVEL(e) to process the edges in $\mathcal{E}$ in a non-decreasing order of their levels as follows.

Suppose i is the currently scanned index of H. If $H[i]$ is non-empty, we process edges of $\mathcal{L}(v)$ for each $v \in H[i]$ one by one. Processing of an edge may cause a rerooting event that may result in addition of new edges to $\mathcal{E}$. We insert these edges and their corresponding endpoints in H accordingly. In addition, this rerooting may result in the fall of some vertices already present in H. We move these vertices to their new position in H accordingly. After processing an edge, say $e = (u, v)$, e is removed from $\mathcal{L}(u)$ and $\mathcal{L}(v)$. If $\mathcal{L}(u)$ (likewise $\mathcal{L}(v)$) becomes empty, we remove u (likewise v) from H. As a result, all the vertices in $H[i]$ will be removed and $H[i]$ will become empty. Notice that all the edges collected in $\mathcal{E}$ during any rerooting event that occurs while processing $H[i]$ will be at a level lower than i. This is because all these collected edges will be from the subtree now hanging from the edge that caused the rerooting event. Thus, we just need to continue scanning H in the rightward direction until $\mathcal{E}$ becomes empty.

It follows that the data structure H facilitates processing of the edges in $\mathcal{E}$ in non-decreasing order of their levels. It requires $O(1)$ time for inserting an edge in $\mathcal{E}$ and updating the level of a vertex. The only overhead in maintenance of H is the time required for scanning through the cells of H. This factor becomes significant only when a lot of empty cells have to be scanned in H to find the next highest edge in $\mathcal{E}$. This cost can be charged to the fall of endpoints of edges in $\mathcal{E}$ as follows. Consider any edge $e \in \mathcal{E}$ which is collected during a rerooting event caused by processing some edge, say e_0. The level of edge e may fall several times before it is finally processed and removed from $\mathcal{E}$. The entire cost of scanning through H from LEVEL(e_0) to the eventual level of e when it is finally processed is charged as follows. After the rerooting event caused by processing e_0, the difference between LEVEL(e_0) and LEVEL(e) is always less than the fall of the endpoint of e on the conjugate path. Any subsequent change in LEVEL(e) will be charged to the fall of that endpoint of e which led to this change. Hence, the time spent in scanning through H during the entire algorithm is bounded by the total sum of fall of vertices in T, which is $O(n^2)$.

6.3 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 simply 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 Fig. 4). 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 state the following theorem.

Theorem 5 *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* $O(n^2)$.

The $O(n^2)$ bound of our algorithm is quite tight even for sparse graphs. In fact, in the next section we show the following: There exists a sequence of $\Theta(n)$ edge insertions such that every incremental algorithm for maintaining DFS tree *explicitly* (using parent or child pointers) that follows the principle of *monotonic fall* will require $\Omega(n^2)$ time.

6.4 Tightness of Analysis Even for Sparse Graphs

Consider a graph on n vertices with $n/2 - 3$ isolated vertices in the beginning, and a set P of $n/2$ vertices hanging from a vertex x. Let $par(x) = y$ and $par(y) = z$, and each vertex in P has a back edge to y (see Fig. 5a). Now use 4 isolated vertices t, u, v, w, and connect them using the edges (z, t), (t, u), (u, v), (v, w) followed by insertion of edge (w, x) (see Fig. 5b). If the algorithm follows *monotonic fall*, it must hang y from x (see Fig. 5c) and then eventually hang P from y (see Fig. 5d). This creates a structure similar to the original tree shown in Fig. 5a. This process changes $n/2$ tree edges and hence requires at least $\Omega(n)$ time. This step can be repeated $n/8$ times. Hence, overall inserting $\Theta(n)$ edges will take $\Omega(n^2)$ time. Thus, we have the following theorem.

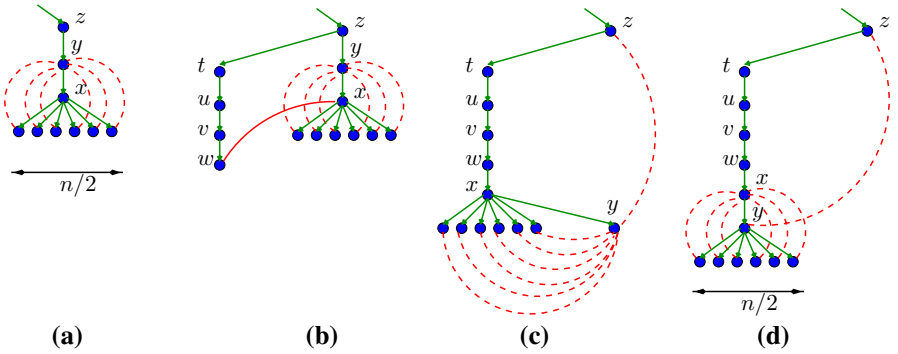

Fig. 5 Example to show tightness of analysis

Theorem 6 *Any incremental algorithm that follows the principle of monotonic fall, will require a total update time of* $\Omega(n^2)$ *for maintaining a DFS Tree explicitly even for sparse graphs.*

7 Conclusion

We presented a simple and efficient algorithm for maintaining a DFS tree for an undirected graph under insertion of edges. It will be interesting to see its counterpart for directed graphs in the future.

References

1. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: Proceedings of Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, 9–15 July 2000, pp. 73–84 (2000)
2. Baswana, S., Choudhary, K.: On dynamic DFS tree in directed graphs. In: Proceedings of Part II Mathematical Foundations of Computer Science 2015—40th International Symposium, MFCS 2015, Milan, Italy, 24–28 Aug 2015, pp. 102–114 (2015)
3. Baswana, S., Khan, S.: Incremental algorithm for maintaining DFS tree for undirected graphs. In: Proceedings of Part I Automata, Languages, and Programming—41st International Colloquium, ICALP 2014, Copenhagen, Denmark, 8–11 July 2014, , pp. 138–149 (2014)
4. Baswana, S., Khurana, S., Sarkar, S.: Fully dynamic randomized algorithms for graph spanners. ACM Trans. Algorithms **8**(4), 35 (2012)
5. Cole, R., Hariharan, R.: Dynamic lca queries on trees. SIAM J. Comput. **34**(4), 894–923 (2005)
6. Demetrescu, C., Italiano, G.F.: A new approach to dynamic all pairs shortest paths. J. ACM **51**(6), 968–992 (2004)
7. Eppstein, D., Galil, Z., Italiano, G.F., Nissenzweig, A.: Sparsification—a technique for speeding up dynamic graph algorithms. J. ACM **44**(5), 669–696 (1997)
8. Erdős, P., Rényi, A.: On the evolution of random graphs. In: Publications of the Mathematical Institute of the Hungarian Academy of Sciences, vol. 5, pp. 17–61 (1960)
9. Franciosa, P.G., Gambosi, G., Nanni, U.: The incremental maintenance of a depth-first-search tree in directed acyclic graphs. Inf. Process. Lett. **61**(2), 113–120 (1997)
10. Gottlieb, L., Roditty, L.: Improved algorithms for fully dynamic geometric spanners and geometric routing. In: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, vol. 2008 2008, pp. 591–600 (2008)

11. Henzinger, M.R., King, V.: Randomized fully dynamic graph algorithms with polylogarithmic time per operation. J. ACM **46**(4), 502–516 (1999)
12. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J. ACM **48**(4), 723–760 (2001)
13. Kapron, B.M., King, V., Mountjoy, B.: Dynamic graph connectivity in polylogarithmic worst case time. In: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, 6–8 January 2013, pp. 1131–1142 (2013)
14. Krivelevich, M., Sudakov, B.: The phase transition in random graphs: a simple proof. Random Struct. Algorithms **43**(2), 131–138 (2013)
15. Miltersen, P.B., Subramanian, S., Vitter, J.S., Tamassia, R.: Complexity models for incremental computation. Theor. Comput. Sci. **130**(1), 203–236 (1994)
16. Reif, J.H.: Depth-first search is inherently sequential. Inf. Process. Lett. **20**(5), 229–234 (1985)
17. Reif, J.H.: A topological approach to dynamic graph connectivity. Inf. Process. Lett. **25**(1), 65–70 (1987)
18. Roditty, L.: Fully dynamic geometric spanners. Algorithmica **62**(3–4), 1073–1087 (2012)
19. Roditty, L., Zwick, U.: Improved dynamic reachability algorithms for directed graphs. SIAM J. Comput. **37**(5), 1455–1471 (2008)
20. Roditty, L., Zwick, U.: Dynamic approximate all-pairs shortest paths in undirected graphs. SIAM J. Comput. **41**(3), 670–683 (2012)
21. Sankowski, P.: Dynamic transitive closure via dynamic matrix inverse (extended abstract). In: Proceedings of 45th Symposium on Foundations of Computer Science (FOCS 2004), pp. 509–517 (2004)
22. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
23. Thorup, M.: Fully-dynamic min-cut. Combinatorica **27**(1), 91–127 (2007)