Fully Dynamic Depth-First Search in Directed Graphs

Bohua Yang[♮], Dong Wen[♮], Lu Qin[♮], Ying Zhang[♮], Xubo Wang[♮], and Xuemin Lin[§]
[♮]Centre for Artificial Intelligence, University of Technology Sydney, Australia
[§]The University of New South Wales, Australia
[♮]bohua.yang@student.uts.edu.au; {dong.wen, lu.qin, ying.zhang, xubo.wang}@uts.edu.au;
[§]lxue@cse.unsw.edu.au;

ABSTRACT

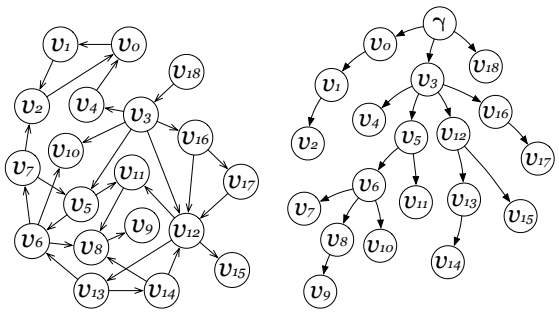
Depth-first search (DFS) is a fundamental and important algorithm in graph analysis. It is the basis of many graph algorithms such as computing strongly connected components, testing planarity, and detecting biconnected components. The result of a DFS is normally shown as a DFS-Tree. Given the frequent updates in many real-world graphs (e.g., social networks and communication networks), we study the problem of DFS-Tree maintenance in dynamic directed graphs. In the literature, most works focus on the DFS-Tree maintenance problem in undirected graphs and directed acyclic graphs. However, their methods cannot easily be applied in the case of general directed graphs. Motivated by this, we propose a framework and corresponding algorithms for both edge insertion and deletion in general directed graphs. We further give several optimizations to speed up the algorithms. We conduct extensive experiments on 12 real-world datasets to show the efficiency of our proposed algorithms.

PVLDB Reference Format:
Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Xubo Wang and Xuemin Lin. Fully Dynamic Depth-First Search in Directed Graphs. *PVLDB*, 13(2): 142-154, 2019.
DOI: https://doi.org/10.14778/3364324.3364329

1. INTRODUCTION

Depth-first search (DFS)[1] is an algorithm to traverse a graph. It searches the vertices along a graph as far as possible in each branch before backtracking. The process of a DFS is naturally represented as a search spanning tree following the depth-first order, named the DFS-Tree. Given a graph G in Figure 1(a), one possible DFS-Tree $\mathcal{T}$ of G is shown in Figure 1(b). The time complexity for performing a DFS traversal and generating a DFS-Tree in a graph $G(V, E)$ is $O(|V| + |E|)$ [20].

DFS is a fundamental algorithm in graph analysis and is the basis for efficiently solving numerous graph problems, such as testing graph reachability [19,24], detecting strongly connected components [9,17,20], detecting biconnected components [11], finding graph bridges [21], finding paths, detecting cycles [23], testing bipartiteness, testing graph planarity [8,12], and topological sorting [22]. These algorithms perform DFS traversal as a subroutine. They require access to vertices in the depth-first order.



(a) The graph G (b) A DFS-Tree $\mathcal{T}$ of G

Figure 1: An example graph G and its DFS-Tree $\mathcal{T}$. γ is a virtual root connecting all vertices in G.

In many real-world applications, graphs dynamically update over time. Given the importance of DFS, the DFS-Tree maintenance problem in dynamic directed graphs is insufficiently studied. In this paper, we examine this problem, which is to update the DFS-Tree for an inserted or deleted edge. The aforementioned applications of DFS benefit from this study. Specifically, in many graph problems such as computing strongly connected components [20], biconnected components [11], and finding graph bridges [21], a key step is to compute the reachable ancestor with the lowest depth of each vertex in the DFS-Tree. Based on this study, we can simply finish this task by directly tracking the updated DFS-Tree of the graph. For example, in detecting biconnected components, it is required to compute a DFS-Tree of the graph and then traverse the tree to get the shallowest position that each vertex can reach. When the graph updates, we can derive an updated DFS-Tree instead of performing the DFS traversal from scratch. We can also simultaneously maintain the interval label (discovery time and finish time) of each vertex as a byproduct in the DFS-Tree. The interval label is used in several works [19, 24] as a part of the index to test the graph reachability. These works filter out the queries if two vertices are connected in the tree, and it only takes constant time to check the reachability in the tree using the interval labels. Based on the study in this paper, we can immediately derive the updated interval labels when the graph updates instead of rerunning DFS. In addition, in

[1] https://en.wikipedia.org/wiki/Depth-first_search

Proceedings of the VLDB Endowment, Vol. 13, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3364324.3364329

puzzle problems such as mazes, users can check the updated interval labels to efficiently identify the connectivity from the entrance to the goal when the maze updates. They can also directly search the updated DFS-Tree to find a solution.

Existing Works and Challenges. The DFS-Tree maintenance problem for directed acyclic graphs and undirected graphs has been well studied in the literature. However, these techniques cannot be applied to the DFS-Tree maintenance of general directed graphs.

For directed acyclic graphs, [10] and [2] investigate the DFS-Tree maintenance problem under incremental settings and decremental settings, respectively. Franciosa et al. [10] update the DFS-Tree by locating a range of vertices according to the postorder of the DFS-Tree. They only reconstruct the tree structure for the located range of vertices. Correctness is guaranteed by the property of directed acyclic graphs that there is no backward edge in its DFS-Tree. However, if we follow the same procedure as that in [10] in general directed graphs, a backward edge starting from a vertex in the located range and ending at a vertex outside the range may become a forward-cross edge after the update is finished. Here, an edge (s, t) is a forward-cross edge if s is visited before t in the preorder of the tree, and there is no ancestor-descendant relationship between s and t. A tree with any forward-cross edge is not a valid DFS-Tree. Considering a deleted tree edge (u, v), the decremental algorithm proposed by Baswana and Choudhary [2] iteratively finds a new position for each vertex in the subtree of v following the topological order. The property of directed acyclic graphs guarantees that the in-neighbors of a vertex do not contain its descendants in the DFS-Tree, whereas in general directed graphs, a vertex may have a descendant as a potential parent which cannot be appended back to the tree.

Baswana et al. [1, 4] and Chen et al. [5, 6] propose fully dynamic algorithms to maintain the DFS-Tree in undirected graphs. They partition the DFS-Tree into disjoint subtrees and paths. The property of undirected graphs guarantees that there is no cross edge between subtrees, and the neighbor of a vertex appears either as its ancestor or its descendant in the DFS-Tree. However, these properties are not applicable to directed graphs since a directed graph may have cross edges in its DFS-Tree, and two adjacent vertices do not always have ancestor-descendant relationships.

Baswana et al. [3] design an incremental algorithm to maintain a DFS-Tree in general directed graphs based on the algorithm presented in [10]. They make use of a structure called stick, which is a long downward path from the root on which there is no branching after a large number of edge insertions [3]. However, the stick structure may be broken due to the edge deletion. Therefore, their algorithm cannot be easily used in the fully dynamic setting. Motivated by the above limitations, we propose efficient, easy-to-implement, and fully dynamic algorithms for DFS-Tree maintenance in general directed graphs.

Our Solution. Given a graph G and its DFS-Tree $\mathcal{T}$, it is necessary to update the DFS-Tree $\mathcal{T}$ if a forward-cross edge has been inserted or a tree edge has been deleted. We use the time intervals to efficiently check the edge type in constant time, where the time interval of a vertex u is an interval starting from the discovery timestamp and ending at the finish timestamp of u in DFS. For the clarity of presentation, we add a virtual root γ connecting all vertices in the graph, so there always exists a DFS-Tree for any graph. An edge removal operation for edge (s, t) can be transformed into appending the subtree rooted at t to end of the children list of the virtual root γ, and this step may generate several new forward-cross edges due to the back movement of the subtree. Therefore, the tree update is essentially to eliminate the forward-cross edges for both edge insertion and deletion. Instead of naively reconstructing the whole DFS-Tree, we first propose a general framework for the tree update based on the concept of time interval. The key step in the framework is to set a range called candidate interval. The candidate interval locates a small set of vertices. We replace the DFS-subtree induced by these vertices by performing DFS only for these vertices in the new graph, whereas the other part of the DFS-Tree remains unchanged. We give the implementations for both edge insertion and deletion. By carefully setting the candidate interval, the computed DFS-Tree is guaranteed to be valid.

To improve the algorithmic efficiency of the basic implementation, we propose several optimizations for both edge insertion and deletion from two perspectives. First, we aim to refine the candidate interval and reduce the number of influenced vertices. Instead of using a fixed candidate interval, we adopt a new strategy that dynamically updates the candidate interval during the process of DFS. We guarantee that the search scope is at most the same as that in the basic algorithm in the worst case. Second, we transform a part of the graph search to the tree search. Recall that in the basic implementation, we scan the out-neighbors of all located vertices in DFS to collect their children in the updated DFS-Tree. We observe that the (tree) children of a set of vertices in the old DFS-Tree can be reused in the updated DFS-Tree, so we avoid scanning the out-neighbors of these vertices in the graph. Our experiments show that the proportion of this kind of vertices is very large, and this optimization greatly speeds up the algorithm especially in large graphs with many high-degree vertices.

Contributions. We summarize the main contributions in this paper as follows.

- *A general and flexible framework.* We design a novel framework for both edge insertion and deletion. To the best of our knowledge, we are the first to study the fully dynamic DFS-Tree maintenance problem in general directed graphs from the perspective of practical implementation.
- *Easy-to-implement algorithms.* We develop algorithms based on the proposed framework for both operations. The algorithms are easy to implement in practice.
- *Two groups of optimizations.* We optimize the algorithms for both operations in two directions. One is to tighten the candidate interval. This reduces the search scope and guarantees that the running time of algorithms only depends on the neighbors of vertices whose visiting time has been changed in the updated DFS-Tree. The other one is to scan the children in the DFS-Tree instead of the out-neighbors in the graph for a large proportion of visited vertices. This optimization further improves the algorithmic efficiency.
- *Extensive experiments.* We conduct experiments on 12 real-world networks to show the performance of our proposed algorithms and the effectiveness of our optimizations.

Table 1: Notations

Notation	Description
$N_{in}(u)$	the in-neighbors of vertex u
$N_{out}(u)$	the out-neighbors of vertex u
$\mathcal{C}(u)$	the children list of vertex u in the DFS-Tree $\mathcal{T}$
$\mathcal{T}(u)$	the vertex set in the subtree rooted at u in $\mathcal{T}$
$\mathcal{I}_{\mathcal{T}}(u)$	the time interval of u in $\mathcal{T}$
$\mathcal{T}[t]$	the visited vertex at the timestamp t in $\mathcal{T}$
$\mathcal{T}[l,r]$	the visited vertex set in the interval $[l,r]$ in $\mathcal{T}$
$\mathcal{T}(\mathcal{I})$	equivalent to $\mathcal{T}[\mathcal{I}.left, \mathcal{I}.right]$
$\mathcal{T}_{new}$	the updated DFS-Tree

Outline. The rest of this paper is organized as follows. Section 2 introduces background knowledge about DFS and defines the research problem. Section 3 introduces related works. Section 4 gives a framework for DFS-Tree maintenance. Section 5 gives the basic implementation for both edge insertion and deletion. Section 6 studies the optimizations. Section 7 reports the experiment result, and Section 8 concludes the paper. Due to the space limitation, we omit the detailed proof for some lemmas and theorems if they are extremely straightforward.

2. PRELIMINARY

We study a directed graph $G(V, E)$, where V is the set of vertices and E is the set of edges in G. The number of vertices and edges are denoted by n and m respectively, i.e., $n = |V|$ and $m = |E|$. Given a vertex u, we denote the in-neighbors (resp. out-neighbors) of u by $N_{in}(u)$ (resp. $N_{out}(u)$), and denote the in-degree (resp. out-degree) of u by $d_{in}(u) = |N_{in}(u)|$ (resp. $d_{out}(u) = |N_{out}(u)|$). Several frequently used notations are summarized in Table 1.

Definition 1. (Depth-First Search) *Given a graph G, a depth-first search (DFS) traverses G in a particular order by picking an unvisited vertex v from the out-neighbors of the most recently visited vertex u to search, and backtracks to the vertex from where it came when a vertex u has explored all possible ways to search further. [7]*

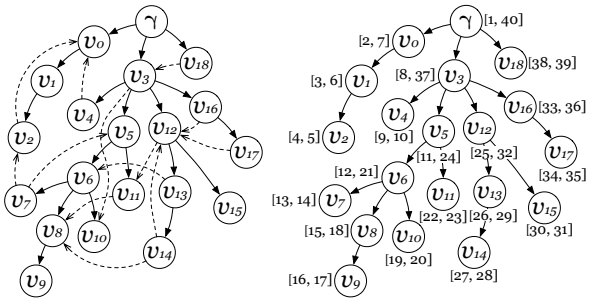
For simplicity and without loss of generality, we add a virtual root vertex γ and connect γ to every vertex in G. We always perform the DFS traversal starting from γ and collect all vertices.

Definition 2. (DFS-Tree) *Given a graph G, a DFS-Tree of G, denoted by $\mathcal{T}_G$, is an ordered spanning tree formed by the process of DFS. [7]*

Algorithm 1: DFS(u)

```
Input: a graph G, and a root vertex u in G
Output: a DFS-Tree T
1 mark u as visited;
2 foreach v ∈ N_out(u): v is unvisited do
3 |   append v to the end of children list C(u) in T;
4 |   DFS(v);
```

We omit the subscript G of $\mathcal{T}_G$ when the context is clear. Given a vertex u, we denote $\mathcal{C}(u)$ the children list of u in the DFS-Tree $\mathcal{T}$. Note that the DFS-Tree is not unique. There is a one-to-one correspondence between a vertex search order and a DFS-Tree. We give the pseudocode for computing the DFS-Tree in Algorithm 1, which is self-explanatory. Given an example graph G in Figure 1(a), one possible DFS-Tree $\mathcal{T}$ of graph G is shown in Figure 1(b).



(a) Types of non-tree edges (b) Time intervals

Figure 2: The non-tree edges and time intervals of the DFS-Tree $\mathcal{T}$.

The Validity of the DFS-Tree. Given a graph G and any search spanning tree $\mathcal{T}$ of G, the edges appearing in the tree are called *tree edges*. The remaining edges (u, v) are called *non-tree edges* and are categorized into one the following four types:

- (u, v) is a *forward edge* if u is an ancestor of v in $\mathcal{T}$.
- (u, v) is a *backward edge* if u is a descendant of v in $\mathcal{T}$.
- (u, v) is a *forward-cross edge* if u and v do not have an ancestor/descendant relationship, and u is visited before v in the preorder of $\mathcal{T}$.
- (u, v) is a *backward-cross edge* if u and v do not have an ancestor/descendant relationship, and u is visited after v in the preorder of $\mathcal{T}$.

Example 1. *We show the non-tree edges for the DFS-Tree $\mathcal{T}$ of Figure 1(b) in Figure 2(a). The edge (v_3, v_{10}) is a forward edge since v_3 is the ancestor of v_{10}. (v_2, v_0) is a backward edge since v_2 is a descendant of v_0. (v_7, v_2) is a backward-cross edge since v_7 is visited after v_2, and these two vertices do not have an ancestor/descendant relationship. There is no forward-cross edge in $\mathcal{T}$.*

Lemma 1. *Given a graph G, a search spanning tree of G is a DFS-Tree if and only if there is no forward-cross edge in G under this tree. [18]*

Problem Definition. In this paper, we study the problem of maintaining the DFS-Tree in dynamic directed graphs. Formally, given a directed graph G and a DFS-Tree $\mathcal{T}$ of G, we aim to efficiently compute a search spanning tree of G without any forward-cross edge when an edge is inserted into or deleted from G.

Note that we only focus on the edge operation since the vertex update can be implemented by several edge updates.

3. RELATED WORK

Reif [15] shows that the ordered DFS problem is a P-complete problem. Here, the ordered DFS problem traverses the graph according to the order specified by the adjacency lists, and the ordered DFS-Tree is unique. Reif [16] and Miltersen et al. [13] prove that the P-completeness of a problem implies hardness of the problem in a dynamic environment. In addition to the hardness of the ordered DFS-Tree maintenance problem, in the literature, the DFS-Tree maintenance problem in directed acyclic graphs [2, 10] and undirected graphs [1, 4–6] has been well studied. However, as discussed in Section 1, these techniques cannot be applied to the fully dynamic DFS-Tree maintenance of general directed graphs.

For directed acyclic graphs, Franciosa et al. [10] propose an incremental algorithm to maintain a DFS-Tree under a sequence of edge insertions in $O(mn)$ total time. Baswana and Choudhary [2] propose a randomized decremental algorithm to maintain a DFS-Tree under a sequence of edge deletions with expected $O(mn \log n)$ total time. Baswana et al. [3] extend the incremental algorithm for directed acyclic graphs presented in [10] to general directed graphs.

For undirected graphs, Baswana et al. [1] propose a fully dynamic algorithm for maintaining a DFS-Tree under a sequence of updates with $O(\sqrt{mn}\log^{2.5} n)$ time per update, an incremental algorithm for maintaining a DFS-Tree under a sequence of edge insertions with $O(n \log^3 n)$ time per edge insertion, and a fault-tolerant algorithm for computing a DFS-Tree of graph $G \setminus \mathcal{F}$ with $O(nk \log^4 n)$ time under any set $\mathcal{F}$ of k failed vertices or edges [1]. The time complexities of the above algorithms are further improved by Chen et al. [5, 6] to $O(\sqrt{mn}\log^{1.5} n)$ for the fully dynamic algorithm, $O(n)$ for the incremental algorithm, and $O(nk \log^2 n)$ for the fault-tolerant algorithm. Nakamura and Sadakane [14] optimize the space occupied by the data structure in the above algorithms from $O(m \log^2 n)$ to $O(m \log n)$. Moreover, Baswana et al. [4] improve the above algorithms to achieve $O(\sqrt{mn \log n})$ time for the fully dynamic algorithm and $O(n(k' + \log n) \log n)$ time for the fault-tolerant algorithm, where k' is the maximum number of failed vertices/edges along any root-leaf path of the initial DFS-Tree.

4. A FLEXIBLE FRAMEWORK

In this section, we first introduce several important concepts in checking the validity of the DFS-Tree, then present a framework for the DFS maintenance problem.

4.1 Efficient Validity Check

As mentioned in the previous sections, a valid DFS-Tree does not contain any forward-cross edge. Note that for a deleted tree edge (s, t), we can first connect the tree by appending t to the end of the children list of the visual root γ. Several new forward-cross edges may appear as a result. Therefore, handling the edge deletion can also be transformed to the problem of eliminating froward-cross edges. Given an edge (s, t), we can efficiently check the edge type using the *time interval* of each vertex instead of scanning the out-neighbors (resp. in-neighbors) of s (resp. t) in the graph.

Definition 3. (Time Interval) *Given a DFS-Tree $\mathcal{T}$ and a vertex u, the time interval of u is denoted as $\mathcal{I}_\mathcal{T}(u) = [x, y]$, where $x = \mathcal{I}_\mathcal{T}(u).left$ is the discovery timestamp of u in the DFS traversal, and $y = \mathcal{I}_\mathcal{T}(u).right$ is the finish timestamp of u when its out-neighbors have been examined completely in the DFS traversal. [7]*

Corollary 1. *There exists a one-to-one correspondence between the DFS-Tree and the time interval of each vertex.*

We omit the subscript $\mathcal{T}$ of $\mathcal{I}_\mathcal{T}$ when the context is clear. We give the time interval of every vertex in the DFS-Tree $\mathcal{T}$ of Figure 1(b) in Figure 2(b). In the rest of this paper, we always use the term *discovery* u and *finish* u to represent the timestamp $\mathcal{I}(u).left$ and $\mathcal{I}(u).right$ respectively. The term *visit* u means either discovery u or finish u.

Given a timestamp t ($1 \le t \le 2n$) and the DFS-Tree $\mathcal{T}$, we denote $\mathcal{T}[t]$ the visited vertex at timestamp t, i.e., $\mathcal{T}[t] = v$ iff $\mathcal{I}(v).left = t \vee \mathcal{I}(v).right = t$. For example, $\mathcal{T}[12] = v_6$ in Figure 2(b). Given a time interval $\mathcal{I}$, we denote $\mathcal{T}(\mathcal{I})$ (or $\mathcal{T}[\mathcal{I}.left, \mathcal{I}.right]$) the visited vertex set no earlier than the timestamp $\mathcal{I}.left$ and no later than the timestamp $\mathcal{I}.right$, i.e., $\mathcal{T}(\mathcal{I}) = \{v \in V | \mathcal{I}.left \le \mathcal{I}(v).left \le \mathcal{I}.right \vee \mathcal{I}.left \le \mathcal{I}(v).right \le \mathcal{I}.right\}$. For example, $\mathcal{T}[8, 15] = \{v_3, v_4, v_5, v_6, v_7, v_8\}$ in Figure 2(b).

We always use the term *search spanning tree* to denote the tree structure that may contain forward-cross edges in the rest. Based on the concept of the time interval, a non-tree edge (s, t) in a search spanning tree is

- a forward edge if $\mathcal{I}(t) \subset \mathcal{I}(s)$,
- a backward edge if $\mathcal{I}(s) \subset \mathcal{I}(t)$,
- a forward-cross edge if $\mathcal{I}(s).right < \mathcal{I}(t).left$, or
- a backward-cross edge if $\mathcal{I}(t).right < \mathcal{I}(s).left$.

To efficiently check the edge types, we maintain the time interval of each vertex in addition to the tree structure.

4.2 The Framework

To eliminate the forward-cross edges in a search spanning tree, the general idea of our framework is to locate a candidate part of the tree, then reconstruct the tree structure and recompute the time intervals of this part. We propose a one-pass strategy. By one-pass, we mean to simultaneously update the tree structure and the time interval of each visited vertex. In addition, we also propose optimizations that dynamically refine the range in Section 6. The pseudocode of the framework is given in Algorithm 2.

Algorithm 2: DFS-Maintenance Framework

Input: a directed graph G, a search spanning tree $\mathcal{T}$ with forward-cross edges
Output: the updated DFS-Tree

```
1  set a candidate time interval CI;
2  r ← LCA(T[CI.left], T[CI.right]);
3  ts ← CI.left;
4  ConstrainedDFS(r);
5  return the updated DFS-Tree T;
```

Algorithm 3: ConstrainedDFS(u)

```
1  mark u as visited;
2  if I(u).left ≥ CI.left then
3      I(u).left ← ts, ts ← ts + 1;
4  foreach v ∈ N_out(u): I(v) ∩ CI ≠ ∅ ∧ I(v) ⊉ CI ∧ v
     is unvisited do
5      if I(v).left ≥ CI.left then
6          v' ← T[ts − 1];
7          if v' = u then
8              reassign v to the first element in C(u);
9          else
10             reassign v to the next element of v' in C(u);
11     ConstrainedDFS(v);
12 if I(u).right ≤ CI.right then
13     I(u).right ← ts, ts ← ts + 1;
```

We locate a part of the DFS-Tree by setting a candidate time interval $\mathcal{CI}$ (line 1). Let r be the lowest common ancestor (LCA) of the first visited vertex $\mathcal{T}[\mathcal{CI}.left]$ and the last visited vertex $\mathcal{T}[\mathcal{CI}.right]$ (line 2). We perform a constrained DFS starting from r (line 4). Here, by constrained, we mean only to visit the vertices falling in the candidate time interval $\mathcal{CI}$ during the DFS.

The pseudocode of ConstrainedDFS is given in Algorithm 3. Note that the initial state of every vertex in the graph is unvisited in all the algorithms proposed in this paper. The variables $\mathcal{CI}$ and ts are global variables. We update the discovery time of u if the original discovery time of u falls in the interval $\mathcal{CI}$ (lines 2–3). Then, we recursively discover the out-neighbor v of u if v falls in $\mathcal{CI}$ (lines 4–11). Note that in line 4, $\mathcal{I}(v) \cap \mathcal{CI} \neq \emptyset \wedge \mathcal{I}(v) \not\supset \mathcal{CI}$ is equivalent to $v \in \mathcal{T}(\mathcal{CI})$. In line 7, $v' = u$ means that v is the first discovered child after discovering u. Otherwise, v' is the last finished child of u before v. We recursively search the constrained neighbors of v by invoking ConstrainedDFS(v) in line 11. Finally, we update the finish time of u if its original finish time falls in the interval $\mathcal{CI}$ (lines 12–13).

4.3 Framework Analysis

Correctness Analysis. We prove the correctness of Algorithm 2 in this subsection. We use $\mathcal{T}$ to denote the input search spanning tree in Algorithm 2. We add the subscript new in the tree notation (shown as $\mathcal{T}_{new}$) when necessary for clarity to represent the updated search spanning tree returned by Algorithm 2.

LEMMA 2. *Given any time interval $\mathcal{CI}$ in a search spanning tree $\mathcal{T}$, the lowest common ancestor of all vertices visited during $\mathcal{CI}$ is the same as that of the first and the last visited vertices during $\mathcal{CI}$, i.e.,* $\mathsf{LCA}(\mathcal{T}[\mathcal{CI}.left], \mathcal{T}[\mathcal{CI}.right]) = \mathsf{LCA}(\mathcal{T}(\mathcal{CI}))$.

PROOF. Let $u = \mathcal{T}[\mathcal{CI}.left]$, $v = \mathcal{T}[\mathcal{CI}.right]$, w be the LCA of u and v, and w' be the LCA of $\mathcal{T}(\mathcal{CI})$. Since both w and w' are the ancestor of u and all the ancestors of u are on the simple path from u to the root, either w and w' have an ancestor-descendant relationship or they are the same vertex. To derive $w' = w$, we show w is not the ancestor of w' and vice versa.

First, w is not the ancestor of w'. Otherwise, the LCA of u and v would be w'. Second, we prove that w is not the descendant of w' by contradiction. Assume that w is the descendant of w'. There is a vertex u' in $\mathcal{T}(\mathcal{CI})$ such that u' is not the descendant of w. Then, either u' is the ancestor of w, or u' and w do not have an ancestor-descendant relationship. For the first case, we have $\mathcal{I}(u') \supset \mathcal{CI}$, and for the second case, we have $\mathcal{I}(u') \cap \mathcal{CI} = \emptyset$ since $\mathcal{I}(u') \cap \mathcal{I}(w) = \emptyset$ and $\mathcal{I}(w) \supseteq \mathcal{CI}$. Both two cases contradict that u' is in $\mathcal{T}(\mathcal{CI})$. □

LEMMA 3. *For each vertex $v \in \mathcal{T}(\mathcal{CI})$ in Algorithm 2, either $v = r$ or there is a tree path from r to v such that each vertex in the path (excluding r) is in $\mathcal{T}(\mathcal{CI})$.*

PROOF. We prove it by contradiction. Assume that there is a vertex $v \in \mathcal{T}(\mathcal{CI})$ and $v \neq r$, and the parent v' of v satisfies $v' \neq r$ and $v' \notin \mathcal{T}(\mathcal{CI})$. Then, either $\mathcal{I}(v') \cap \mathcal{CI} = \emptyset$ or $\mathcal{I}(v') \supset \mathcal{CI}$.

For the first case, we have $\mathcal{I}(v) \cap \mathcal{CI} = \emptyset$ because $\mathcal{I}(v) \subset \mathcal{I}(v')$. This contradicts that $v \in \mathcal{T}(\mathcal{CI})$. For the second case, r is the LCA of all vertices in $\mathcal{T}(\mathcal{CI})$ based on Lemma 2. However, if $\mathcal{I}(v') \supset \mathcal{CI}$, we hold that v' is the LCA of all vertices in $\mathcal{T}(\mathcal{CI})$, since v' is the common ancestor of $\mathcal{T}[\mathcal{CI}.left]$ and $\mathcal{T}[\mathcal{CI}.right]$, and v' is a descendant of r. This contradicts that r is the LCA of all vertices in $\mathcal{T}(\mathcal{CI})$. □

Based on the above two lemmas, there is an invocation for ConstrainedDFS() for each vertex $v \in \mathcal{T}(\mathcal{CI})$, and we do not lose any vertex belonging to $\mathcal{T}(\mathcal{CI})$ in the constrained DFS. On the other hand, given the limitation in line 4 of Algorithm 3, we guarantee that if a vertex v is visited in the original tree during the interval $\mathcal{CI}$, v will also be visited in the updated tree during $\mathcal{CI}$. A formal lemma is given as follows.

LEMMA 4. *For each vertex v in Algorithm 2, $\mathcal{I}_{\mathcal{T}_{new}}(v).left$ (resp. $\mathcal{I}_{\mathcal{T}_{new}}(v).right$) falls in $\mathcal{CI}$ if and only if $\mathcal{I}_{\mathcal{T}}(v).left$ (resp. $\mathcal{I}_{\mathcal{T}}(v).right$) falls in $\mathcal{CI}$, i.e., $\mathcal{T}(\mathcal{CI}) = \mathcal{T}_{new}(\mathcal{CI})$.*

THEOREM 1. *Given an input search spanning tree $\mathcal{T}$ of G, Algorithm 2 computes a valid DFS-Tree if (i) there is no forward-cross edge (u, v) in $\mathcal{T}$ such that $u \in V \setminus \mathcal{T}(\mathcal{CI})$; and (ii) there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI}) \wedge \mathcal{I}(v).left > \mathcal{CI}.right$.*

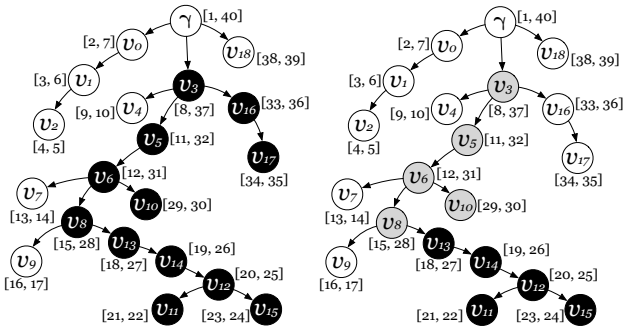
PROOF. The updated search spanning tree $\mathcal{T}_{new}$ computed by Algorithm 2 is a DFS-Tree if and only if there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$. Two vertices $u, v \in V$ can be considered in the following four cases: (i) $u \in \mathcal{T}(\mathcal{CI}) \wedge v \in \mathcal{T}(\mathcal{CI})$, (ii) $u \in \mathcal{T}(\mathcal{CI}) \wedge v \in V \setminus \mathcal{T}(\mathcal{CI})$, (iii) $u \in V \setminus \mathcal{T}(\mathcal{CI}) \wedge v \in \mathcal{T}(\mathcal{CI})$, and (iv) $u \in V \setminus \mathcal{T}(\mathcal{CI}) \wedge v \in V \setminus \mathcal{T}(\mathcal{CI})$.

For the case (i), Algorithm 3 guarantees that there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI}) \wedge v \in \mathcal{T}(\mathcal{CI})$. Because if the edge (u, v) exists, v would be visited in the invocation of ConstrainedDFS(u).

For the case (ii), the condition (ii) in the theorem guarantees that there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI}) \wedge v \in V \setminus \mathcal{T}(\mathcal{CI})$. If $\mathcal{I}(v).left \leq \mathcal{CI}.right$, either $\mathcal{I}(v).right < \mathcal{CI}.left$ or $\mathcal{I}(v) \supset \mathcal{CI}$, since $v \notin \mathcal{T}(\mathcal{CI})$. For both two cases, $\mathcal{I}(v).left \leq \mathcal{CI}.left$, considering that $\mathcal{I}(u).right \geq \mathcal{CI}.left$ since $u \in \mathcal{T}(\mathcal{CI})$, the edge (u, v) cannot be a forward-cross edge since $\mathcal{I}(u).right \geq \mathcal{I}(v).left$.

For the case (iii), the condition (i) in the theorem guarantees that there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in V \setminus \mathcal{T}(\mathcal{CI}) \wedge v \in \mathcal{T}(\mathcal{CI})$. On the one hand, if $\mathcal{I}(u).left > \mathcal{CI}.right$ or $\mathcal{I}(u) \supset \mathcal{CI}$, we will have $\mathcal{I}(u).right \geq \mathcal{CI}.right \geq \mathcal{I}(v).left$. Therefore, the edge (u, v) cannot be a forward-cross edge. On the other hand, if $\mathcal{I}(u).right < \mathcal{CI}.left$, we assume that there is a forward-cross edge (u, v) in $\mathcal{T}_{new}$, then edge (u, v) in $\mathcal{T}$ may be a tree edge, forward edge, backward edge, forward-cross edge, or backward-cross edge. Firstly, edge (u, v) cannot be a tree edge or forward edge in $\mathcal{T}$. If so, we would have $\mathcal{I}(u) \supset \mathcal{I}(v)$, and this contradicts $\mathcal{I}(u).right < \mathcal{CI}.left$. Secondly, edge (u, v) cannot be a backward edge in $\mathcal{T}$. If so, it would be a backward edge in $\mathcal{T}_{new}$ too. Specifically, $u \in \mathcal{T}(r)$, and u is visited before $\mathcal{T}[\mathcal{CI}.left]$ during DFS. Following Algorithm 3, we visit vertices in the preorder of $\mathcal{T}$ before visiting $\mathcal{T}[\mathcal{CI}.left]$. Therefore, v is an ancestor of u in $\mathcal{T}_{new}$, and edge (u, v) is a backward edge. Thirdly, edge (u, v) cannot be a backward-cross edge in $\mathcal{T}$. If so, we would have $\mathcal{I}(v).right < \mathcal{I}(u).left < \mathcal{I}(u).right < \mathcal{CI}.left$, and this contradicts $v \in \mathcal{T}(\mathcal{CI})$. Considering aforementioned conditions, edge (u, v) must be a forward-cross edge in $\mathcal{T}$. We have that if edge (u, v) is not a forward-cross edge in $\mathcal{T}$, it will not be a forward-cross edge in $\mathcal{T}_{new}$ under the condition $\mathcal{I}(u).right < \mathcal{CI}.left$.

For the case (iv), the condition (i) in the theorem guarantees that there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in V \setminus \mathcal{T}(\mathcal{CI}) \wedge v \in V \setminus \mathcal{T}(\mathcal{CI})$. Since the time interval of a vertex not in $\mathcal{T}(\mathcal{CI})$ will not change during Algorithm 2, if $\mathcal{I}(u).right \geq \mathcal{I}(v).left$ in $\mathcal{T}$, we will have $\mathcal{I}(u).right \geq \mathcal{I}(v).left$ in $\mathcal{T}_{new}$ too. □



(a) Result of Algorithm 4 (b) Result of Algorithm 6

Figure 3: The updated DFS-Tree for the inserted edge (v_8, v_{13}) in the graph G.

THEOREM 2. *Given a directed graph G and a search spanning tree $\mathcal{T}$ of G, the running time of Algorithm 2 is bounded by $O(\sum_{u \in \mathcal{T}(\mathcal{CI}) \cup \{r\}} d_{out}(u))$, where r is the LCA of all vertices in $\mathcal{T}(\mathcal{CI})$.*

5. IMPLEMENTATIONS

5.1 Edge Insertion

We give the basic implementation for the edge insertion in this subsection. The pseudocode is shown in Algorithm 4.

Algorithm 4: DFS-Insert

Input: a directed graph G, a DFS-Tree $\mathcal{T}$ of G and an inserted edge (s, t) in G
Output: the updated DFS-Tree

```
1 insert (s,t) into G;
2 if I(s).right > I(t).left then return T;
3 r ← LCA(s,t);
4 CI ← [I(s).right, I(r).right];
5 ts ← CI.left;
6 ConstrainedDFS(r);
7 return the updated DFS-Tree T;
```

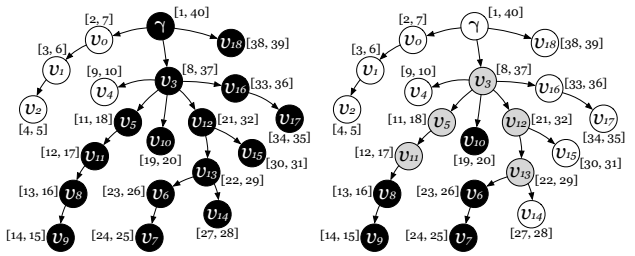
We do nothing and return the original tree if (s, t) is not a forward-cross edge in line 2. We compute the LCA of s and t as the search root of ConstrainedDFS() in line 3 and set the candidate interval as $[\mathcal{I}(s).right, \mathcal{I}(r).right]$ in line 4. ConstrainedDFS() is invoked in line 6. A running example is given as follows.

EXAMPLE 2. *Given the directed graph G in Figure 1(a) and its DFS-Tree $\mathcal{T}$ in Figure 1(b), the updated DFS-Tree $\mathcal{T}_{new}$ computed by Algorithm 4 for an inserted edge (v_8, v_{13}) is shown in Figure 3(a). The LCA of v_8 and v_{13} in the original DFS-Tree $\mathcal{T}$ is v_3. v_3 is also the LCA of all black vertices, which is supported by Lemma 2. According to Figure 2(b), the candidate time interval $\mathcal{CI}$ is assigned by $[18, 37]$. The vertices falling in the candidate time interval $\mathcal{CI}$ are marked in black. The time intervals of vertices which do not belong to $\mathcal{T}(\mathcal{CI})$ (white vertices) do not change in the algorithm.*

LEMMA 5. *In Algorithm 4, there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI})$ and $\mathcal{I}(v).left > \mathcal{CI}.right$.*

The correctness of Algorithm 4 is guaranteed by combining Theorem 1 and Lemma 5. We give the time complexity of Algorithm 4 as follows.

THEOREM 3. *Given a directed graph G, a DFS-Tree $\mathcal{T}$ of G and an inserted edge (s, t), the running time of Algorithm 4 is bounded by $O(\sum_{u \in \mathcal{T}[\mathcal{I}(s).right, \mathcal{I}(r).right]} d_{out}(u))$, where r is the LCA of s and t.*



(a) Result of Algorithm 5 (b) Result of Algorithm 8

Figure 4: The updated DFS-Tree for the deleted edge (v_5, v_6) in the graph G.

5.2 Edge Deletion

We explain the basic implementation for the edge deletion in this subsection. The pseudocode is shown in Algorithm 5. The pseudocode is self-explanatory, so we omit the detailed description here.

Algorithm 5: DFS-Delete

Input: a directed graph G, a DFS-Tree $\mathcal{T}$ of G and a deleted edge (s, t) in G
Output: the updated DFS-Tree

```
1 delete (s,t) from G;
2 if (s,t) is not a tree edge then return T;
3 γ ← the virtual root of the DFS-Tree T;
4 CI ← [I(t).left, I(γ).right];
5 ts ← CI.left;
6 ConstrainedDFS(γ);
7 return the updated DFS-Tree T;
```

EXAMPLE 3. *Given the directed graph G in Figure 1(a) and its DFS-Tree $\mathcal{T}$ in Figure 1(b), the updated DFS-Tree $\mathcal{T}_{new}$ computed by Algorithm 5 for a deleted edge (v_5, v_6) is presented in Figure 4(a). According to Figure 2(b), the candidate time interval $\mathcal{CI}$ is assigned by $[12, 40]$. Similar to Figure 3(a), the vertices falling in the candidate time interval $\mathcal{CI}$ are marked in black, and the time intervals of vertices which do not belong to $\mathcal{T}(\mathcal{CI})$ (white vertices) do not change in the algorithm.*

LEMMA 6. *In Algorithm 5, there is no forward-cross edge (u, v) in $\mathcal{T}_{new}$ such that $u \in \mathcal{T}(\mathcal{CI})$ and $\mathcal{I}(v).left > \mathcal{CI}.right$.*

The correctness of Algorithm 5 is guaranteed by combining Theorem 1 and Lemma 6. We give the time complexity of Algorithm 5 as follows.

THEOREM 4. *Given a directed graph G, a DFS-Tree $\mathcal{T}$ of G and a deleted edge (s, t), the running time of Algorithm 5 is bounded by $O(\sum_{u \in \mathcal{T}[\mathcal{I}(t).left, \mathcal{I}(\gamma).right]} d_{out}(u))$, where γ is the virtual root of $\mathcal{T}$.*

6. THE IMPROVED APPROACHES

Drawbacks of Basic Solutions. Even though Algorithm 4 and Algorithm 5 correctly update the DFS-Tree, there is still much room for improvement. First, the key step in both Algorithm 4 and Algorithm 5 is to set a candidate time interval $\mathcal{CI}$. This interval can be very large, and many vertices will consequently be visited in ConstrainedDFS(). Second, all the out-neighbors of each discovered vertex will be scanned in ConstrainedDFS(). The total number of their out-neighbors can be very large especially in big graphs.

We propose several optimizations to improve the algorithmic efficiency. In response to the drawbacks of the basic solutions, we first adopt a strategy which dynamically refines the candidate time interval $\mathcal{CI}$. Specifically, we continuously adjust the candidate interval in the process of the algorithm, and the candidate interval is theoretically guaranteed to be never larger than that of the basic solutions. In addition, we design a hybrid approach to perform the constrained DFS. By hybrid, we mean searching vertices by combining the graph search and the tree search. This avoids scanning all the out-neighbors of the visited vertices in the basic solutions. We introduce the details for edge insertion and deletion in Section 6.1 and Section 6.2 respectively.

6.1 Edge Insertion

6.1.1 Tightening Candidate Interval

We first give some key observations for tightening the candidate interval in the edge insertion algorithm.

LEMMA 7. *Given an inserted edge (s,t), let C be the set of new descendants of s in Algorithm 4, i.e. $C = (\mathcal{T}_{new}(s) \setminus \{s\}) \cap \mathcal{T}(\mathcal{CI})$, we have $C = \mathcal{T}_{new}(t)$.*

LEMMA 8. *Given an inserted edge (s,t), let w be the vertex in $\mathcal{T}_{new}(t)$ with the largest old right interval value, i.e., $w = \arg\max_{v \in \mathcal{T}_{new}(t)} \mathcal{I}(v).right$. There is no edge (u,v) such that $u \in \mathcal{T}_{new}(t)$ and $\mathcal{I}(v).left > \mathcal{I}(w).right$.*

EXAMPLE 4. *Consider the graph G in Figure 1(a), the original DFS-Tree $\mathcal{T}$ in Figure 1(b), and the updated DFS-Tree $\mathcal{T}_{new}$ in Figure 3(a) for an inserted edge (v_8, v_{13}). The new descendants of v_8 is $C = \mathcal{T}_{new}(v_{13}) = \{v_{11}, v_{12}, v_{13}, v_{14}, v_{15}\}$. v_{12} has the largest old right interval value 32 as shown in Figure 2(b). The set of vertices with a left time interval larger than 32 is $\{v_{16}, v_{17}, v_{18}\}$, and there is no edge from the vertex in $\mathcal{T}_{new}(v_{13})$ to the vertex in this set.*

Based on Lemma 8, we can use $[\mathcal{I}(s).right, \mathcal{I}(w).right]$ as the new candidate interval and guarantee the algorithmic correctness. To derive $\mathcal{I}(w).right$, we dynamically update the candidate interval in the process of the edge insertion algorithm. Specifically, given an inserted edge (s,t), we can initialize the candidate interval as $\mathcal{T}[\mathcal{I}(s).right, \mathcal{I}(t).right]$ since t must be in $\mathcal{T}_{new}(t)$. Every time a new vertex v is assigned as a descendant of t, we update $\mathcal{CI}.right$ to $\mathcal{I}(v).right$ if $\mathcal{I}(v).right > \mathcal{CI}.right$. The candidate interval stops updating once all the descendants of t are collected in the updated DFS-Tree $\mathcal{T}_{new}$. The following lemma shows that the search space of this new candidate interval is at most the same as that of Algorithm 4 in the worst case.

LEMMA 9. *Given an inserted edge (s,t), let w be the vertex defined in Lemma 8. $\mathcal{I}(w).right \leq \mathcal{I}(r).right$, where r is the LCA of s and t.*

6.1.2 From Graph Search to Tree Search

We further improve the algorithmic efficiency by replacing a part of the graph search with tree search.

LEMMA 10. *Given an inserted edge (s,t) in Algorithm 4, when finishing the visit of vertex t, i.e., $u = t$ in line 12 of Algorithm 3, there is no forward-cross edge in the graph.*

Based on Lemma 10, at the finish moment of vertex t, the tree is a valid DFS-Tree, and what we need is to update the time intervals of the remaining vertices. Therefore, we can simply search the tree instead of the graph to finish the update. We give an example to explain Lemma 10.

EXAMPLE 5. *We continue to use Figure 3(a) for the inserted edge (v_8, v_{13}). When all the descendants of v_{13} are collected (the timestamp is 27), there is no forward-cross edge. At this moment, all remaining vertices still in the candidate interval are $\{v_3, v_5, v_6, v_8, v_{10}\}$. Compared with Figure 2(b), the tree structures of these remaining vertices do not change, and we only update their time intervals.*

6.1.3 The Overall Algorithm

By combining these two optimization techniques mentioned above, we detail our final algorithm for edge insertion in this subsection. The pseudocode is presented in Algorithm 6.

Algorithm 6: DFS-Insert*

Input: a directed graph G, a DFS-Tree $\mathcal{T}$ of G and an inserted edge (s,t) in G
Output: the updated DFS-Tree

```
1  insert (s,t) into G;
2  if I(s).right > I(t).left then return T;
3  r ← LCA(s,t);
4  CI ← [I(s).right, I(t).right];
5  reassign t to the last element in C(s);
6  ts ← CI.left;
7  InsHybrid-DFS(r,t);
8  return the updated DFS-Tree T;
```

Algorithm 7: InsHybrid-DFS(u,t)

```
1  mark u as visited;
2  if I(u).left ≥ CI.left then
3      I(v).left ← ts, ts ← ts + 1;
4  if u ∈ T(t) then
       // Graph Search
5      C(u) = ∅;
6      foreach v ∈ N_out(u):
         I(v) ∩ CI ≠ ∅ ∧ I(v) ⊅ CI ∧ v is unvisited do
7          reassign v to the last element in C(u);
8          CI.right ← max(CI.right, I(v).right);
9          InsHybrid-DFS(v,t);
10 else
       // Tree Search
11     foreach v ∈ C(u): I(v) ∩ CI ≠ ∅ do
12         InsHybrid-DFS(v,t);
13 if I(u).right ≤ CI.right then
14     I(u).right ← ts, ts ← ts + 1;
```

In line 4 of Algorithm 6, we initialize the candidate interval as $[\mathcal{I}(s).right, \mathcal{I}(t).right]$. We invoke the subroutine named InsHybrid-DFS instead of ConstrainedDFS, and the search root is still the LCA of s and t.

The pseudocode of InsHybrid-DFS is presented in Algorithm 7. Similar to Algorithm 3, we update the left time interval of u in lines 2–3 and the right time interval of u in lines 13–14 respectively if necessary. We check whether u is in $\mathcal{T}(t)$ in line 4. If yes, we perform the graph search in lines 5–9. The constraint (line 6) in the graph search is set to be the same as that in line 4 of Algorithm 3. In line 8, we update the right bound of the candidate interval if necessary. If u is not in $\mathcal{T}(t)$, we perform the tree search in lines 11–12. Line 11 guarantees that only the vertices falling in the candidate interval are visited. We give a running example of Algorithm 6 as follows.

Example 6. *Given directed graph G in Figure 1(a) and its DFS-Tree $\mathcal{T}$ in Figure 1(b), the updated DFS-Tree $\mathcal{T}_{new}$ computed by Algorithm 6 for the inserted edge (v_8, v_{13}) is shown in Figure 3(b). For each visited vertex, if a graph search is performed, the vertex is marked in black, or if a tree search is performed, the vertex is marked in gray. Consider the tree in Figure 3(a) derived by Algorithm 4. The candidate interval is* $[18, 37]$*, and there are* 12 *(black) vertices visited in the algorithm, whereas in Figure 3(b), the final stable candidate interval is* $[18, 32]$*. Only* 10 *(black and gray) vertices are visited in the algorithm, and we only visit the tree children instead of the graph out-neighbors of the* 5 *(gray) vertices inside.*

Theorem 5. *Given a directed graph G, a DFS-Tree $\mathcal{T}$ of G and an inserted edge (s, t), the running time of Algorithm 6 is $O(|\mathcal{T}[\mathcal{I}(s).right, \mathcal{I}(w).right]| + \sum_{u \in \mathcal{T}_{new}(t)} d_{out}(u))$, where w is the vertex defined in Lemma 8.*

The left part of the complexity is the number of visited vertices in the candidate interval $[\mathcal{I}(s).right, \mathcal{I}(w).right]$, and the right part is the number of out-neighbors of all vertices in $\mathcal{T}_{new}(t)$. In brief, the left and right part represent the tree search and graph search in Algorithm 6 respectively. Based on Lemma 9, the overall time complexity of Algorithm 6 is not larger than that of Algorithm 4.

6.2 Edge Deletion

In this subsection, we propose the optimizations for updating the DFS-Tree when a tree edge is deleted.

6.2.1 A Similar Optimization to Edge Insertion

Recall that given a deleted tree edge (s, t) in Algorithm 5, we straightforwardly append the subtree $\mathcal{T}(t)$ to the end of the children list of the virtual root γ. This essentially transforms the edge deletion problem to a forward-cross edge repairing problem. This method is inefficient due to the unpredictable generated forward-cross edges. A wide candidate interval (from $\mathcal{I}(t).left$ to $\mathcal{I}(\gamma).right$) is set to cover the possibly influenced vertices.

In order to tighten the candidate interval, one method is to adopt a similar optimization in Section 6.1.1, which dynamically updates the candidate interval. Specifically, we start the constrained DFS from the timestamp $\mathcal{I}(t).left$, i.e., $\mathcal{CI}.left = \mathcal{I}(t).left$, which is the same as that in Algorithm 5, and do not limit the right bound of the candidate interval initially. The DFS terminates immediately once all the vertices in $\mathcal{T}(t)$ are visited.

However, the improvement of this optimization is limited since the new ancestors of vertices in $\mathcal{T}(t)$ are unpredictable, and we need to scan all the out-neighbors of every vertex visited in the DFS.

6.2.2 Avoiding Unpredictable Graph Search

To improve the efficiency of the edge deletion, we adopt a new method that iteratively appends the vertices in $\mathcal{T}(t)$ to the tree as early as possible. For the vertices not in $\mathcal{T}(t)$, the tree search is performed and the time interval is updated in the tree. After visiting a special kind of vertex u which can be the parent of vertices in $\mathcal{T}(t)$, we start the graph search and collect a set of vertices in $\mathcal{T}(t)$ as the descendants of u. The algorithm terminates once all vertices in $\mathcal{T}(t)$ are appended to the tree. This new method further tightens the candidate interval, and the graph search is only performed on the vertices in $\mathcal{T}(t)$. We introduce the details below and start by giving several definitions for ease of understanding.

Definition 4. (Local Earliest Visit Time) *Given a vertex $u \in \mathcal{T}(t)$ and a vertex $p \in N_{in}(u) \cap (V \setminus \mathcal{T}(t))$, the local earliest visit time of u regarding p, denoted by $\mathcal{EV}_p(u)$, is the smallest integer i such that $i > \mathcal{I}(p).left$ and $\nexists v \in \mathcal{C}(p) : \mathcal{I}(v).left < \mathcal{I}(t).left \wedge i \leq \mathcal{I}(v).right$.*

Definition 5. (Earliest Visit Time and Potential Parent) *Given a vertex $u \in \mathcal{T}(t)$, the earliest visit time of u, denoted by $\mathcal{EV}(u)$, is the smallest local earliest visit time of u, i.e., $\mathcal{EV}(u) = \min_{p \in N_{in}(u) \cap (V \setminus \mathcal{T}(t))} \mathcal{EV}_p(u)$. The corresponding vertex p of the minimum $\mathcal{EV}_p(u)$ is the potential parent of u, denoted by $\mathcal{PP}(u)$.*

Note that due to the existence of virtual root γ, every vertex in $\mathcal{T}(t)$ has a potential parent and earliest visit time. The condition $i > \mathcal{I}(p).left$ guarantees that we visit u as a child of p. Given for any other child $v \in \mathcal{C}(p)$, $\mathcal{I}(u) \cap \mathcal{I}(v) = \emptyset$. The condition $\nexists v \in \mathcal{C}(p) : \mathcal{I}(v).left < \mathcal{I}(t).left \wedge i \leq \mathcal{I}(v).right$ guarantees that i is the earliest position to append u to $\mathcal{C}(p)$ after the timestamp $\mathcal{I}(t).left$. We give an example as follows.

Example 7. *Reconsider the case of deleting the edge (v_5, v_6) in the DFS-Tree shown in Figure 2. The vertex set in the subtree $\mathcal{T}(v_6)$ is $\{v_6, v_7, v_8, v_9, v_{10}\}$. The vertex v_8 has three in-neighbors $\{v_{11}, v_{14}, \gamma\}$ that are not in $\mathcal{T}(v_6)$. The local earliest visit time $\mathcal{EV}_{v_{11}}(v_8)$ is* 23*, $\mathcal{EV}_{v_{14}}(v_8)$ is* 28*, and $\mathcal{EV}_{\gamma}(v_8)$ is* 38*. Therefore, the potential parent of v_8 is v_{11} and its earliest visit time is* 23*. In other words, the earliest position to append v_8 after the timestamp $\mathcal{I}(v_6).left = 12$ in the tree is the child of v_{11}.*

The vertex v_{10} has two in-neighbors $\{v_3, \gamma\}$ that are not in $\mathcal{T}(v_6)$. The local earliest visit time $\mathcal{EV}_{v_3}(v_{10})$ is 25*, and $\mathcal{EV}_{\gamma}(v_{10})$ is* 38*. Thus, the potential parent of vertex v_{10} is v_3 and its earliest visit time is* 25*. In this case, v_{10} can be appended to $\mathcal{C}(v_3)$ after v_5, since v_5 has been visited before $\mathcal{I}(v_6).left = 12$, and $\mathcal{EV}(v_{10})$ should be larger than $\mathcal{I}(v_5).right = 24$.*

Definition 6. (Trigger) *Given a vertex set $C \subseteq \mathcal{T}(t)$, the trigger of C is a vertex u with the smallest earliest visit time in C, i.e., $u = \arg\min_{v \in C} \mathcal{EV}(v)$.*

Note that it is possible that there are multiple triggers. We break the tie by randomly selecting one in the algorithm. An example to explain the above definitions is given below.

Example 8. *Given the set $C = \mathcal{T}(v_6)$, the trigger is v_8. Its earliest visit time and potential parent is* 23 *and v_{11} respectively. For the other vertices in $\mathcal{T}(v_6)$, we have $\mathcal{PP}(v_{10}) = v_3, \mathcal{EV}(v_{10}) = 25$ and $\mathcal{PP}(v_6) = v_{13}, \mathcal{EV}(v_6) = 27$. The virtual root γ is the only one in-neighbor not in $\mathcal{T}(v_6)$ for the vertices v_7 and v_9. We have $\mathcal{PP}(v_7) = \mathcal{PP}(v_9) = \gamma$ and $\mathcal{EV}(v_7) = \mathcal{EV}(v_9) = 38$.*

We explain the rationale behind these definitions. Firstly, we only allow the vertices in $\mathcal{T}(t)$ to append to the tree after the timestamp $\mathcal{I}(t).left$. This guarantees that there is no new forward-cross edge (u, v) such that $u \in \mathcal{T}(t)$ and v is visited before $\mathcal{I}(t).left$. Secondly, we find the earliest potential parent for each vertex in $\mathcal{T}(t)$, and this avoids the appearance of forward-cross edges due to the backward movement of vertices in $\mathcal{T}(t)$. We give examples as follows.

Example 9. *Following the previous example, consider vertex v_7. The only in-neighbor of v_7 not in $\mathcal{T}(v_6)$ is γ. According to Definition 4 and Definition 5, the earliest position to append v_7 is after v_3. If we omit the condition $\nexists v \in \mathcal{C}(p) : \mathcal{I}(v).left < \mathcal{I}(t).left \wedge i \leq \mathcal{I}(v).right$ in Definition 4 and append v_7 as the first child of γ, this will generate*

two new forward-cross edges (v_7, v_2) *and* (v_7, v_5)*. Therefore, we only append vertices after the timestamp* $\mathcal{I}(t).left$*.*

On the other hand, consider vertex v_8*. There are three in-neighbors not in* $\mathcal{T}(v_6)$*,* v_{11}, v_{14} *and* γ*.* v_{11} *is the potential parent. If we append* v_8 *to* $\mathcal{C}(v_{14})$*, it will generate a new forward-cross edge* (v_{11}, v_8)*.*

Based on Definition 4, Definition 5 and Definition 6, we compute the first trigger u and its potential parent p by scanning the in-neighbors of $\mathcal{T}(t)$. We compute the earliest visit time of u and append u to the children list of p accordingly. We perform the tree search and update the time intervals starting from the timestamp $\mathcal{I}(t).left$. Once visiting the trigger, we perform the graph search and collect its all descendants. Note that for each trigger, only the vertices in $\mathcal{T}(t)$ are appended to its new subtree. Once all descendants of the trigger are collected, we locate the next trigger and repeat this step. We terminate the procedure once all vertices in $\mathcal{T}(t)$ are visited. The following subsection gives a detailed algorithm and corresponding analysis.

6.2.3 The Overall Algorithm

We give our final algorithm for edge deletion in this subsection. The pseudocode is presented in Algorithm 8.

Algorithm 8: DFS-Delete*

Input: a directed graph G, a DFS-Tree $\mathcal{T}$ of G and a deleted edge (s,t) in G
Output: the updated DFS-Tree

```
1  delete (s,t) from G;
2  if (s,t) is a non-tree edge then return T;
3  delete t from C(s);
4  V_c ← T(t);
5  CI ← [I(t).left, +∞];
6  ts ← CI.left;
7  LocateNextTrigger();
8  while V_c is not empty do DelHybrid-DFS(r);
9  return the updated DFS-Tree T;
```

Algorithm 9: LocateNextTrigger()

```
1  trigger ← get trigger in V_c based on Definition 6;
2  p ← PP(trigger);
3  v ← T[EV(trigger) − 1];
4  if v = p then
5  |  reassign trigger to the first element in C(p);
6  else
7  |  reassign trigger to the next element of v in C(p);
8  if r is undefined then r ← LCA(s,p);
9  else r ← LCA(r,p);
```

We mark all candidate vertices as V_c in line 4. The left bound of the candidate interval is initialized as $\mathcal{I}(t).left$ in line 5, and the search will start from this timestamp. We do not set the right bound since the algorithm will terminate once all vertices in V_c are visited. We compute the trigger in line 7 before performing the search and continuously invoke the DelHybrid-DFS() until V_c is empty. The pseudocode of LocateNextTrigger() and DelHybrid-DFS() are given in Algorithm 9 and Algorithm 10 respectively.

In Algorithm 9, we first compute the trigger in V_c in line 1. Based on Definition 4 and Definition 5, we reassign the trigger to its earliest visit position in the children list of its potential parent from line 2 to line 7. Similar to the previous algorithms, r is the search entrance for the DelHybrid-DFS() and is initialized by the LCA of s and p (line 8). Note that unlike the previous methods, we dynamically update r, since we compute a new trigger in each invocation of Algorithm 9. The search entrance r must update to cover the new trigger and guarantee the correctness (line 9).

Algorithm 10: DelHybrid-DFS(u)

```
1  mark u as visited;
2  if I(u).left ≥ CI.left ∨ u ∈ V_c then
3  |  I(u).left ← ts, ts ← ts + 1;
4  if u ∈ V_c then
   |  // Graph Search
5  |  V_c ← V_c \ {u};
6  |  C(u) = ∅;
7  |  foreach v ∈ N_out(u): v ∈ V_c ∧ v is unvisited do
8  |  |  reassign v to the last element in C(u);
9  |  |  DelHybrid-DFS(v);
10 else
   |  // Tree Search
11 |  foreach v ∈ C(u): I(v) ∩ CI ≠ ∅ ∨ v ∈ V_c do
12 |  |  DelHybrid-DFS(v);
13 |  |  if v = trigger then
14 |  |  |  if V_c is empty then
15 |  |  |  |  CI.right ← ts − 1;
16 |  |  |  else
17 |  |  |  |  LocateNextTrigger();
18 if I(u).right ≤ CI.right then
19 |  I(u).right ← ts, ts ← ts + 1;
20 |  CI.left ← ts;
```

In Algorithm 10, we first update the left time interval of u if u falls in the candidate interval or $u \in V_c$ in lines 2–3. Note that even though $\mathcal{CI}.left$ is initialized by $\mathcal{I}(t).left$ and $\mathcal{I}(u).left \geq \mathcal{CI}.left$ for all $u \in V_c$ holds at the beginning, we update $\mathcal{CI}.left$ in the algorithm. Thus, condition $u \in V_c$ is necessary. The graph search is performed only for the vertices in V_c and is shown in lines 4–9. In line 7, we limit u's out-neighbors to those belonging to V_c. This is because we postpone visiting u and an out-neighbor $v \notin V_c$ must be visited before. Otherwise, the edge (u, v) would be a forward-cross edge. The tree search is shown in lines 11–17. In line 13, if v is a trigger, a set of vertices has been appended to the subtree of v. If V_c is empty, we actually finish updating the DFS-Tree, and the algorithm can terminate immediately. By setting $\mathcal{CI}.right$ as $ts - 1$, no vertex will be further visited. When V_c is not empty, we invoke LocateNextTrigger() to find the next earliest position and complete the DFS-Tree. In line 18, all visited vertices satisfy this condition until we set $\mathcal{CI}.right$ as $ts - 1$. We update the right time interval of u in line 19. Note that line 20 is an important step to avoid visiting the same vertices repeatedly and to guarantee the algorithmic correctness. Specifically, recall that r updates in Algorithm 9. Let r_{old} be the old one, and r_{new} be the updated value. Assume that r_{new} is an ancestor of r_{old}. In the invocation of DelHybrid-DFS(r_{new}), we need to avoid searching the subtree $\mathcal{T}(r_{old})$ since the new intervals of the vertices inside have been allocated in the invocation of DelHybrid-DFS(r_{old}). In this case, $\mathcal{CI}.left$ has been updated to $\mathcal{I}(r_{old}).right + 1$. We give a running example for Algorithm 8 as follows.

EXAMPLE 10. *Given the directed graph G in Figure 1(a) and its DFS-Tree $\mathcal{T}$ in Figure 1(b), the updated DFS-Tree $\mathcal{T}_{new}$ computed by Algorithm 8 for the deleted edge (v_5, v_6) is shown in Figure 4(b). Similar to the example of edge insertion, each visited vertex is marked in black if the graph search has been performed, and each visited vertex is marked in gray if a tree search has been performed. The visited vertex set is the union of the black and gray vertices.*

Figure 4(a) derived by Algorithm 5 visits 16 (black) vertices. Figure 4(b) only visits 10 vertices. The initial left bound of the candidate interval in Algorithm 8 is $\mathcal{I}(v_6).left = 12$, and the right bound when terminating the algorithm is 26. Of these 10 vertices, we visit the tree children instead of the graph out-neighbors for the 5 vertices.

THEOREM 6. *Given a directed graph G, a DFS-Tree $\mathcal{T}$ of G and a deleted edge (s,t), the running time of Algorithm 8 is bounded by $O(|C| + \sum_{u \in \mathcal{T}(t)} d(u))$, where $|C|$ is the number of vertices visited in the candidate interval and $d(u) = d_{in}(u) + d_{out}(u)$.*

6.3 Batch Update

Our proposed algorithms can be easily extended to handle the batch update. Without loss of generality, a batch update can be considered as a group of edge insertions followed by a group of edge deletions.

Edge Insertion. We assume that all the inserted edges are forward-cross edges. For other kinds of edges, we always first add them into the graph since they will never break the validity of the DFS-Tree. We denote the set of inserted forward-cross edges as $B_+ = \{(s_1,t_1),(s_2,t_2),...,(s_b,t_b)\}$. We set $r = \mathsf{LCA}(s_1,t_1,s_2,t_2,...,s_b,t_b)$ and the candidate interval $\mathcal{CI} = \bigcup_{1 \le i \le b} [\mathcal{I}(s_i).right, \mathcal{I}(\mathsf{LCA}(s_i,t_i)).right]$ in the batch update version of Algorithm 4. The optimization of tightening the candidate interval discussed in Section 6.1.1 can also be applied in the batch insertion. Considering Algorithm 6, besides the modification of r in Algorithm 4, we set $\mathcal{CI} = \bigcup_{1 \le i \le b} [\mathcal{I}(s_i).right, \mathcal{I}(t_i).right]$ and perform the graph search in Algorithm 7 when $\exists 1 \le i \le b : u \in \mathcal{T}(t_i)$.

Edge Deletion. Similarly, we assume that all the deleted edges are tree edges. We denote the set of deleted tree edges as $B_- = \{(s_1,t_1),(s_2,t_2),...,(s_b,t_b)\}$. We set $\mathcal{CI} = [\min_{1 \le i \le b} \mathcal{I}(t_i).left, \mathcal{I}(\gamma).right]$ in the batch update version of Algorithm 5. The optimizations discussed in Section 6.2.1 can also be applied in the batch deletion. Considering Algorithm 8, besides the aforementioned modifications, we set $V_c = \bigcup_{1 \le i \le b} \mathcal{T}(t_i)$.

7. EXPERIMENTS

We conducted extensive experiments to evaluate the performance of our proposed algorithms. The algorithms evaluated in the experiments are summarized as follows:

- ReDFS-Insert and ReDFS-Delete: The naive methods are discussed in Section 4. That is, Algorithm 1 is run if a forward-cross edge is inserted or a tree edge is deleted.
- DFS-Insert and DFS-Delete: Algorithm 4 and Algorithm 5.
- DFS-Insert$^+$ and DFS-Delete$^+$: DFS-Insert$^+$ represents Algorithm 4 with the optimization of tightening the candidate interval discussed in Section 6.1.1. DFS-Delete$^+$ represents Algorithm 5 with a similar optimization discussed in Section 6.2.1.
- DFS-Insert* and DFS-Delete*: Algorithm 6 and Algorithm 8.

Table 2: Network Statistics

Dataset	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
CiteSeer	384,413	1,751,463	4.56
Web-Stanford	281,903	2,312,497	8.20
YahooAd	653,260	2,931,708	4.49
ProsperLoan	89,269	3,394,979	38.03
Wiki-Talk	2,394,385	5,021,410	2.10
Web-Google	875,713	5,105,039	5.83
Flickr	2,302,925	33,140,017	14.39
StackOverflow	2,601,977	63,497,050	24.40
LiveJournal	4,847,571	68,993,773	14.23
UK-2002	18,520,486	298,113,762	16.10
Wiki-Link	12,150,976	378,142,420	31.12
Twitter	41,652,230	1,468,365,182	35.25

All algorithms were implemented in C++ and compiled using a g++ compiler at a -O3 optimization level. All the experiments were conducted on a Linux Server with an Intel Xeon 3.1GHz CPU and 128GB 1600MHz ECC DDR3-RAM.

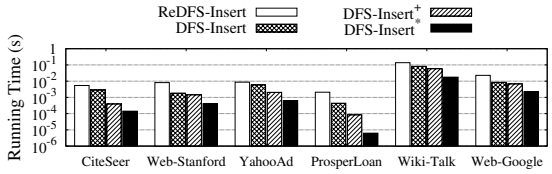
Datasets. We conducted experiments on twelve publicly-available real-world graphs and divided them into two groups according to the data size.

Group one consists of six graphs with a relatively small size. Group two consists of six big graphs. The detailed statistics of these datasets are summarized in Table 2. All networks and corresponding detailed descriptions can be found in SNAP[2], KONECT[3], and WebGraph[4].

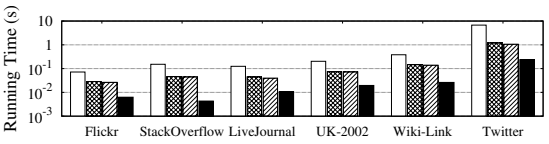
Input Preparation. In order to test the performance of our proposed algorithms, we randomly selected 10000 distinct existing edges in the graph for each test case. For edge deletion, we deleted these 10000 edges from the original graphs one by one. For edge insertion, we started from the graphs after removing these 10000 edges and corresponding DFS-Tree. We inserted them into the graphs one by one. For each algorithm, we calculated the average running time for each edge insertion (resp. deletion).

The rest of this section is summarized as follows. Section 7.1 provides the overall processing time of all the four algorithms. Then in Section 7.2, we evaluate the effectiveness of our proposed optimizations. Section 7.3 evaluates the scalability. Section 7.4 reports the memory usage.

7.1 Overall Efficiency



(a) Small graphs

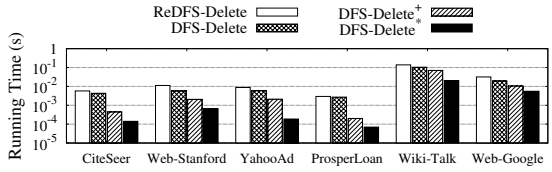


(b) Big graphs

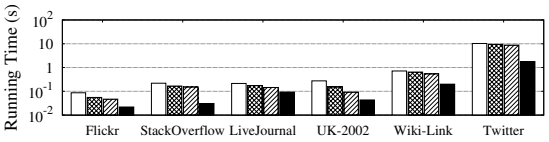
Figure 5: Running time for edge insertion

[2] http://snap.stanford.edu/data/index.html
[3] http://konect.uni-koblenz.de/networks/
[4] http://law.di.unimi.it/datasets.php

Edge Insertions. Figure 5 shows the running time of all the four insertion algorithms on all datasets. Based on our proposed framework, DFS-Insert is more efficient than naively computing the DFS-Tree from scratch (ReDFS-Insert), and our proposed optimization techniques further improve the efficiency of DFS-Insert. For instance, the running time of DFS-Insert* on the smallest graph CiteSeer is $0.14ms$. Meanwhile, the running time of DFS-Insert$^+$, DFS-Insert and ReDFS-Insert is $0.39ms$, $2.84ms$ and $5.40ms$ respectively on the same dataset. On the largest graph Twitter with over 1 billion edges, DFS-Insert* takes around $0.24s$, while DFS-Insert$^+$, DFS-Insert and ReDFS-Insert takes about $1.06s$, $1.19s$ and $6.75s$ respectively.



(a) Small graphs



(b) Big graphs

Figure 6: Running time for edge deletion

Edge Deletions. Figure 6 shows the running time of all the four deletion algorithms on all datasets. Similar to Figure 5, we witness a large improvement from the baseline algorithm to our final algorithm. For example, on the smallest graph CiteSeer, DFS-Delete* takes only $0.14ms$, while DFS-Delete$^+$, DFS-Delete and ReDFS-Delete take $0.44ms$, $4.33ms$ and $5.75ms$ respectively. On the largest graph Twitter, DFS-Delete* takes approximately $1.79s$, while the algorithms DFS-Delete$^+$, DFS-Delete and ReDFS-Delete take around $8.78s$, $9.45s$ and $10.27s$ respectively.

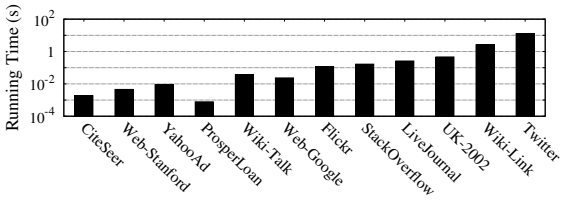
Table 3: Percentage of forward-cross edge insertions

CiteSeer	Web-Stanford	YahooAd	ProsperLoan
7.64%	8.85%	6.95%	0.78%
Wiki-Talk	Web-Google	Flickr	StackOverflow
47.41%	9.55%	5.36%	2.61%
LiveJournal	UK-2002	Wiki-Link	Twitter
4.07%	4.24%	1.01%	1.89%

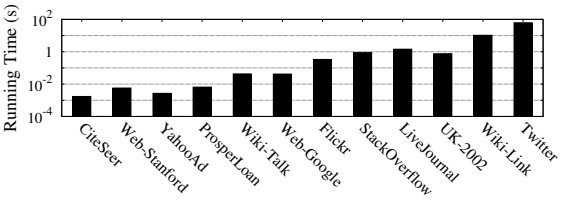
Table 4: Percentage of tree edge deletions

CiteSeer	Web-Stanford	YahooAd	ProsperLoan
8.14%	11.5%	6.95%	1.07%
Wiki-Talk	Web-Google	Flickr	StackOverflow
47.73%	13.3%	6.52%	3.45%
LiveJournal	UK-2002	Wiki-Link	Twitter
6.46%	5.74%	1.92%	2.88%

Update Distribution. Recall that it only takes constant time if the inserted edge is not a forward-cross edge or the deleted edge is not a tree edge. To clearly show the performance of our final algorithm, we report the percentage of the forward-cross edge insertions (DFS-Insert*) and the percentage of the tree edge deletions (DFS-Delete*) in Table 3 and Table 4 respectively. We also report the average running time of DFS-Insert* and DFS-Delete* for these updates in Figure 7 (a) and Figure 7(b) respectively.



(a) Edge insertions

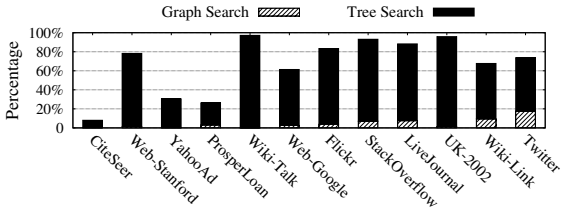


(b) Edge deletions

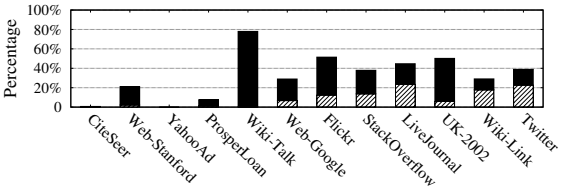
Figure 7: Running time for tree updates

7.2 Effectiveness of Optimizations

We further evaluate the effectiveness of our optimizations. We count the number of ConstrainedDFS() invocations in Algorithm 4 (resp. Algorithm 5) and denote it as c_n. This value represents the number of vertices performing graph search in Algorithm 4 (resp. Algorithm 5). We also count the numbers of vertices by graph search and tree search in our final algorithm respectively, which are denoted by c_g and c_t. Specifically, for each invocation of InsHybrid-DFS() in DFS-Insert*(), we increase c_g by one if it is true in line 4 of InsHybrid-DFS(). Otherwise, we increase c_t by one. Similarly, for edge deletion, we increase c_g by one if it is true in line 4 of DelHybrid-DFS(). Otherwise, we increase c_t by one.



(a) Edge insertions



(b) Edge deletions

Figure 8: Percentage of vertices performing graph search or tree search

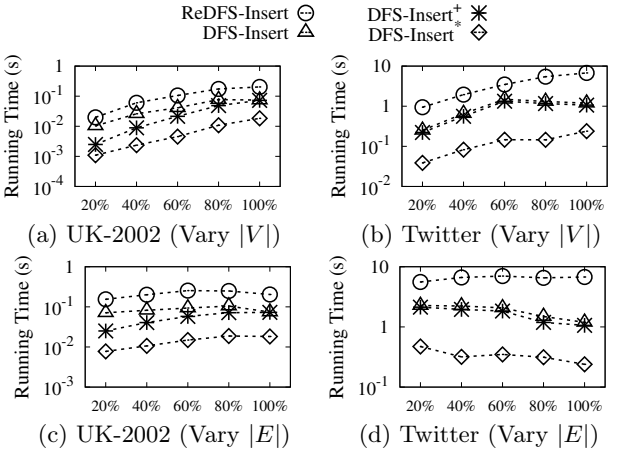


Figure 9: Scalability for edge insertion

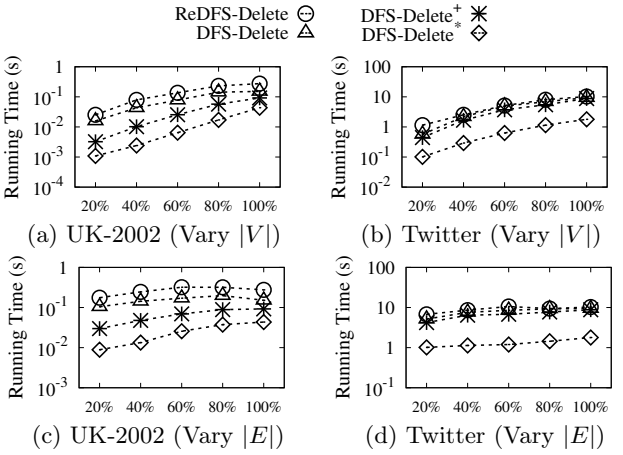


Figure 10: Scalability for edge deletion

We report the value of c_t/c_n and c_g/c_n for both edge insertion and deletion on all datasets in Figure 8. The black part represents the ratio of visited vertices by the tree search c_t/c_n, and the gray part represents the ratio of visited vertices by the graph search c_g/c_n.

The results show that in our final algorithms, the total number of visited vertices is reduced on all datasets, and there is a large proportion of tree search inside. Compared with the basic algorithms for both operations, we only need a very small number of graph searches. For example, the largest percentage of graph search (i.e., c_g/c_n) for edge insertion and deletion is about 17% and 23% on Twitter and LiveJournal respectively. The lowest percentage is 0.000002% on YahooAd for both edge insertion and deletion.

Overall, it is shown that our proposed optimization techniques not only reduce the overall visited vertices but also significantly replace the out-neighbor search by the tree children search.

7.3 Scalability Test

This experiment tests the scalability of our proposed algorithms. Due to the space limitation, we only report the results for two real-world graph datasets as representatives — UK-2002 and Twitter. The results on the other datasets show similar trends. For each dataset, we vary the graph size and graph density by randomly sampling vertices and edges from 20% to 100%. When sampling vertices, we derive the induced subgraph of the sampled vertices, and when sampling edges, we select the incident vertices of the edges as the vertex set.

The running time of DFS-Insert* and DFS-Delete* for edge insertion and deletion of different percentages is reported in Figure 9 and Figure 10, respectively, with other algorithms used as comparisons.

It can be seen that DFS-Insert* and DFS-Delete* perform better than the other algorithms in all cases. For example, considering edge insertion on UK-2002, the running time of DFS-Insert*, DFS-Insert+, DFS-Insert and ReDFS-Insert is $1.09ms$, $2.48ms$, $10.29ms$ and $19.79ms$ respectively when sampling 20% vertices. When the sampling percentage of vertices reaches 100%, DFS-Insert*, DFS-Insert+, DFS-Insert and ReDFS-Insert take about $18.30ms$, $72.70ms$, $74.26ms$ and $203.22ms$ respectively.

Note that in Figure 9, the running time of several algorithms on Twitter sightly decreases when the sampling ratio increases in some cases. This is because in the 10000 update operations, we do not need to update the tree structure if an inserted edge is not a forward-cross edge or a deleted edge is not a tree edge. The proportion of the operations that lead to the tree update may be low in a large graph, so the average processing time for each operation may be small.

7.4 Memory Usage

Figure 11 shows the memory usage of all the four algorithms on all datasets. We do not use any auxiliary data structure in the final algorithm. Therefore, the memory usage of all the proposed algorithms is same.

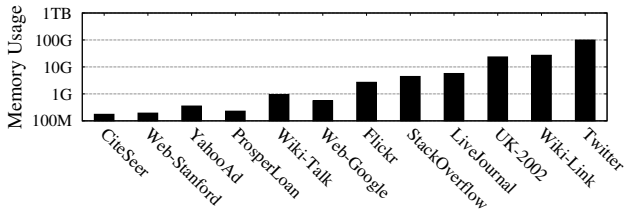


Figure 11: Memory usage

8. CONCLUSIONS

This paper introduces a framework and corresponding algorithms for the DFS-Tree maintenance problem considering both edge insertion and deletion in general directed graphs. Two groups of optimizations are also presented to speed up the algorithms. The results of extensive performance studies demonstrate the efficiency of our proposed algorithms. Based on the studies in this work, several possible research problems have opened. We have given a basic idea to handle the batch update in Section 6.3. A possible direction is to further improve the efficiency of the batch update. Another direction is to explore a vertex order and a corresponding DFS-Tree to reduce the update cost.

9. ACKNOWLEDGEMENTS

Lu Qin is supported by ARC DP160101513. Ying Zhang is supported by ARC FT170100128 and DP180103096. Xuemin Lin is supported by 2018YFB1003504, 2019DH0ZX01, NSFC-61232006, ARC DP180103096 and DP170101628.

10. REFERENCES

[1] S. Baswana, S. R. Chaudhury, K. Choudhary, and S. Khan. Dynamic DFS in undirected graphs: breaking the O(m) barrier. In *Proceedings of the twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 730–739. SIAM, 2016.

[2] S. Baswana and K. Choudhary. On dynamic DFS tree in directed graphs. In *International Symposium on Mathematical Foundations of Computer Science*, pages 102–114. Springer, 2015.

[3] S. Baswana, A. Goel, and S. Khan. Incremental DFS algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 53–72. SIAM, 2018.

[4] S. Baswana, S. K. Gupta, and A. Tulsyan. Fault tolerant and fully dynamic DFS in undirected graphs: simple yet efficient. *arXiv preprint arXiv:1810.01726*, 2018.

[5] L. Chen, R. Duan, R. Wang, and H. Zhang. Improved algorithms for maintaining DFS tree in undirected graphs. *CoRR, abs/1607.04913*, 2016.

[6] L. Chen, R. Duan, R. Wang, H. Zhang, and T. Zhang. An improved algorithm for incremental DFS tree in undirected graphs. In *16th Scandinavian Symposium and Workshops on Algorithm Theory*, 2018.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[8] H. De Fraysseix, P. O. De Mendez, and P. Rosenstiehl. Trémaux trees and planarity. *International Journal of Foundations of Computer Science*, 17(05):1017–1029, 2006.

[9] E. W. Dijkstra. A discipline of programming. *Prentice-Hall series in automatic computation*, 1976.

[10] P. G. Franciosa, G. Gambosi, and U. Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Information processing letters*, 61(2):113–120, 1997.

[11] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[12] J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[13] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203–236, 1994.

[14] K. Nakamura and K. Sadakane. A space-efficient algorithm for the dynamic DFS problem in undirected graphs. In *International Workshop on Algorithms and Computation*, pages 295–307. Springer, 2017.

[15] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[16] J. H. Reif. A topological approach to dynamic graph connectivity. *Information Processing Letters*, 25(1):65–70, 1987.

[17] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[18] J. F. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 282–292. ACM, 2002.

[19] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: can it be even faster? *TKDE*, 29(3):683–697, 2017.

[20] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[21] R. E. Tarjan. A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2:160–161, 1974.

[22] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[23] A. Tucker. Chapter 2: covering circuits and graph colorings. *Applied Combinatorics*, 49, 2006.

[24] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.