Constant-time Connectivity Querying in Dynamic Graphs

LANTIAN XU, University of Technology Sydney, Australia
DONG WEN[*], University of New South Wales, Australia
LU QIN, University of Technology Sydney, Australia
RONGHUA LI, Beijing Institute of Technology, China
YING ZHANG, University of Technology Sydney, Australia
XUEMIN LIN, Shanghai Jiaotong University, China

Connectivity query processing is a fundamental problem in graph processing. Given an undirected graph and two query vertices, the problem aims to identify whether they are connected via a path. Given frequent edge updates in real graph applications, in this paper, we study connectivity query processing in fully dynamic graphs, where edges are frequently inserted or deleted. A recent solution, called D-tree, maintains a spanning tree for each connected component and applies several heuristics to reduce the depth of the tree. To improve the efficiency, we propose a new spanning-tree-based solution by maintaining a disjoint-set tree simultaneously. By combining the advantages of two trees, we achieve the constant query time complexity and also significantly improve the theoretical running time in both edge insertion and edge deletion. Our performance studies on real large datasets show considerable improvement of our algorithms.

CCS Concepts: • **Information systems → Data structures**.

Additional Key Words and Phrases: Connectivity, Connected Component, Dynamic Graph

ACM Reference Format:
Lantian Xu, Dong Wen, Lu Qin, Ronghua Li, Ying Zhang, and Xuemin Lin. 2024. Constant-time Connectivity Querying in Dynamic Graphs. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 230 (December 2024), 23 pages. https://doi.org/10.1145/3698805

1 Introduction

Given an undirected graph, the connectivity query is a fundamental problem and aims to answer whether two vertices are connected via a path. The connectivity query usually serves as a fundamental operator and is mainly used to prune search space in most applications. For example, the algorithms to compute paths between two vertices are generally time-consuming. We can terminate the search immediately if two query vertices are not connected, which avoids unnecessary computation. The problem is driven by a wide range of applications in various fields. For example, the problem helps in identifying whether data packets from a device can reach their destination in

[*]Corresponding author.

Authors' Contact Information: Lantian Xu, University of Technology Sydney, Sydney, Australia, lantian.xu@student.uts.edu.au; Dong Wen, University of New South Wales, Sydney, Australia, dong.wen@unsw.edu.au; Lu Qin, University of Technology Sydney, Sydney, Australia, lu.qin@uts.edu.au; Ronghua Li, Beijing Institute of Technology, Beijing, China, lironghuabit@126.com; Ying Zhang, University of Technology Sydney, Sydney, Australia, ying.zhang@uts.edu.au; Xuemin Lin, Shanghai Jiaotong University, Shanghai, China, xuemin.lin@gmail.com.

ACM 2836-6573/2024/12-ART230
https://doi.org/10.1145/3698805

Algorithms	**Ours**	**D-Tree**
Query processing	$O(\alpha)$	$O(h)$
Edge insertion	$O(h)$	$O(h \cdot \mathsf{nbr}_{\mathsf{update}})$
Edge deletion	$O(h)$	$O(h^2 \cdot \mathsf{nbr}_{\mathsf{scan}})$

Table 1. Comparing the average complexity of our method with state-of-the-art. α is a small constant ($\alpha < 5$). h is the average vertex depth in the spanning tree. $\mathsf{nbr}_{\mathsf{update}}$ is the time to insert a vertex into neighbors of a vertex or to delete a vertex from neighbors of a vertex. $\mathsf{nbr}_{\mathsf{scan}}$ is the time to scan all neighbors of a vertex.

a communication network. In biology, testing connectivity between two units in a protein-protein-interaction network can help in understanding the interactions between different components of the system [17]. For example, existing studies [15] trace contacts during the COVID-19 epidemic by modeling an association graph among the high-risk population, the general population, vehicles, public places, and other entities. Two individuals are considered related if there is a path connecting them within a short time window that satisfies certain patterns. Our method can effectively exclude a suspicious vertex pair if they are not connected within the specified time window.

Real-world graphs are highly dynamic where new edges come in and old edges go out. The connectivity between two vertices may change over time. Given the up-to-date snapshot, the connectivity query can be addressed by graph search strategies such as breadth-first search (BFS) and depth-first search (DFS). However, these online search algorithms may scan the whole graph, which is excessively costly for large graphs. Instead of online query processing, a straightforward index-based approach is to maintain each connected component as a spanning tree, where two query vertices are connected if they have the same tree root. We may merge two spanning trees for a new edge insertion or split a spanning tree for an old edge deletion. Index-based methods have been investigated in the literature. [27] maintains an Euler tour of a spanning tree, named ET-tree. [6, 7] improves the ET-tree by terminating early when looking for replacement edges. Holm et al. proposed a new method named HDT [8, 9], which made the complexity of insertion and deletion $O(\log^2 n)$. [10] further reduced the time complexity slightly to $O(\log n(\log\log n)^2)$.

The State of the Art. To improve the connectivity query efficiency practically, a recent work [3], named D-Tree, maintains a spanning tree in dynamic graphs, and the spanning tree is represented by maintaining the parent of each vertex. To test the connectivity of two vertices, we search from each vertex to the root and identify whether their roots are the same. The query time depends on the depth of each query vertex, i.e., the distance from the query vertex to the root of the tree. Their main technical contribution is to develop several heuristics to maintain a spanning tree with a relatively small average depth. For edge insertion, a non-tree edge is a new edge that connects two vertices in the same tree, and a tree edge is a new edge that connects two vertices in different trees. To insert a non-tree edge (u, v), the D-Tree replaces one existing tree edge with the new edge when the depth gap of u and v is over a certain threshold. To insert a tree edge (u, v) into an index, the D-Tree always merges the smaller tree into the bigger tree. Assume that the spanning tree of u is the smaller one. They rotate the tree (i.e., keep the same tree edges but change the parent-child relationship of certain vertices) so that u becomes the new root. Then, the two trees are connectged by assigning u assigned as the child of v. To delete a tree edge (u, v), the spanning tree is immediately split into two trees T_u and T_v for u and v, respectively. Assume that the size of T_u is smaller. The D-Tree picks T_u and searches non-tree neighbors of each vertex in T_u to find non-tree edges that can reconnect the two trees. The D-Tree picks the edge that connects to the shallowest vertex in T_v and reconnects two trees. Nothing needs to be done when deleting a non-tree edge. When scanning vertices in both tree updates and query processing of the D-Tree, they may rotate the tree if the rotation results in a smaller average depth.

Drawbacks of D-Tree. Much improvement space is left for the D-Tree. We summarize the main drawbacks as follows.

(a) *Poor update efficiency.* In D-Tree, many efforts are spent on reducing the average depth in tree updates at a high price. Especially in tree-edge deletion, they search non-tree neighbors of all vertices in a sub-tree to identify the replacement edge. The search space can be very large which is not scalable for big graphs. In addition, the D-Tree maintains complex structures to search the replacement edge. Maintaining them incurs extra costs when trees are rotated.

(b) *Poor query efficiency.* In the D-Tree, query processing needs to find the roots of two query vertices, and the time complexity of query processing is h, where h represents the average depth of vertices. However, in large-scale graphs, the vertex depth in a spanning tree may be very large, and the query efficiency can be very low.

Our Approach. Table 1 gives a quick view of the theoretical running time of D-Tree and our approach. $\mathsf{nbr}_{\mathsf{update}}$ and $\mathsf{nbr}_{\mathsf{scan}}$ depend on the data structure to maintain non-tree neighbors and children. If a balanced binary tree is used, we have $\mathsf{nbr}_{\mathsf{update}} = \log d$ and $\mathsf{nbr}_{\mathsf{scan}} = d$, where d is the average degree. If the hash set is used, $\mathsf{nbr}_{\mathsf{update}}$ can be reduced to a constant value, but $\mathsf{nbr}_{\mathsf{scan}}$ will be $d + b$ where b is the number of buckets for a hash set. By comparison, our approach achieves the almost-constant query time and reduces the time complexity of both insertion and deletion simultaneously.

We start by considering how to improve the query efficiency in theory. Our idea is inspired by the disjoint-set data structure which organizes items in different sets. It offers two operators: Find identifies the representative of a set containing the element, and Union merges two sets. The structure is commonly identified to achieve the amortized constant time complexity for Find and Union, which correspond to querying connectivity and edge insertion in our case, respectively. However, it is hard to handle deletion situations if we only use the disjoint set. When an edge is deleted only based on the disjoint set, it is not able to identify if the connected component is disconnected, and we need to compute the connected component from scratch as a result. To achieve the constant query time complexity while keeping the high efficiency for both edge insertion and deletion, we maintain a spanning tree and a disjoint set simultaneously and combine the advantages of two data structures. The spanning tree implementation in our algorithm is called ID-Tree, and the disjoint set implementation in our algorithm is called DS-Tree.

Spanning Tree Implementation. Our ID-Tree is extended from D-Tree by applying several modifications for higher practical efficiency. Our results show that our improved version is much more efficient than D-Tree in all aspects. We make the following improvements in response to the drawbacks of D-Tree. We apply a new heuristic early-termination technique to derive a better theoretical bound when searching replacement edges for a deleted tree edge. Given a tree T, after deleting a tree edge, a subtree T_1 with a larger size and a subtree T_2 with a smaller size are formed. We find the replacement edge with lowest depth in the small subtree T_2. This allows us to terminate early and improve the efficiency of searching the replacement edge. In addition, we also avoid maintaining non-tree neighbors and children for each vertex which reduce the maintenance overhead but not sacrifice the update efficiency.

Disjoint Set Implementation. To support edge deletion, we implement the children of each tree vertex in the disjoint set as a doubly linked list. We design a set of operators to delete an item from the disjoint set in constant time while keeping the constant time of Find and Union. When a connected component A is disconnected into B and C after deleting an edge, we use our ID-Tree to identify them (assuming $|B| \leq |C|$). In the disjoint set, we remove all vertices of B from A in $O(|B|)$ time and union all vertices of B to create a new connected component. In this way, the time complexity of updating our disjoint set is bounded by that of the spanning tree.

Contributions. We summarize our main contributions as follows.

- *(a) Theoretical almost-constant query efficiency.* We propose a new approach to combine the advantages of spanning tree and disjoint-set tree. The approach achieves amortized constant query time without sacrificing the time complexity of both edge insertion and edge deletion. Our final solution is theoretically more efficient than the state-of-the-art D-Tree in all aspects.
- *(b) Theoretical higher update efficiency.* We propose a new spanning-tree structure for higher updating efficiency in both theory and practice compared with D-Tree. We also propose a new disjoint-set data structure to handle the edge deletion together with the spanning-tree.
- *(c) Outstanding practical performance.* We conduct extensive experiments on sixteen real datasets in various settings. The results demonstrate the higher practical efficiency of our algorithms compared with the state of the art.

2 Preliminary

Given an undirected simple graph $G(V, E)$, V and E denote the set of vertices and the set of edges, respectively. We use n and m to denote the number of vertices and the number of edges, respectively, i.e., $n = |V|, m = |E|$. The neighbors of a vertex u is represented by $N(u)$, i.e., $N(u) = \{v \in V \mid (u, v) \in E\}$. A tree T is a connected graph without any cycle. Given a tree T, $parent(u)$ denotes the parent vertex of a vertex u in the tree. $depth(u)$ denotes the depth of the vertex u, i.e., the number of tree edges from u to the tree root. h denotes the average depth of all vertices in the tree, i.e., $h(T) = \sum_{u \in T} depth(u)/|T|$. Each tree has only one root. Rotating a tree means keeping the same set of tree edges but changes the tree root. As a result, the parent of each vertex from the old root to the new root becomes the child of the vertex. A path $P = \langle v_1, v_2, ..., v_l \rangle$ in G is a sequence of vertices in which each pair of adjacent vertices are connected via an edge, i.e., for all $1 \leq i < l$, $(v_i, v_{i+1}) \in E$. We say two vertices u, v are connected if there exists a path such that u and v are terminals. A connected component (CC for short) in a graph is a maximal subgraph in which every pair of vertices are connected. Therefore, two vertices are connected if they are in the same connected component.

Definition 2.1. (Connectivity Query) Given a graph G and two query vertices, the connectivity query aims to determine whether u and v are connected in G.

Problem Definition. Given a graph G, we aim to develop an index for processing connectivity queries between arbitrary pairs of vertices and maintain the index when a new edge is inserted or an existing edge is deleted.

3 Existing Solutions

3.1 Query Processing in Static Graphs

A straightforward online method for the connectivity query is to perform a bidirectional breath-first search (BFS) or a depth-first-search from a query vertex. Once meeting the other query vertex in the search, we identify that two vertices are connected. The online method for one connectivity query takes $O(m)$ time in the worst case, which is hard to be tolerated in large graphs. To improve the query efficiency, we can index an identifier of the corresponding connected component for every vertex in a static graph, which takes $O(n)$ space. In this way, the connectivity query can be answered by checking the identifier of two query vertices which takes constant time complexity. Compared with static graphs, dealing with fully dynamic graphs for efficient connectivity query processing is much more challenging. For instance, removing an edge may disconnect the connected component. Certain techniques are expected to identify the connectivity of the original connected component and disconnect the component index structure accordingly.

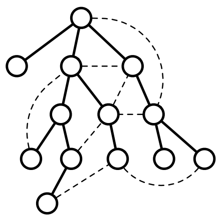

Fig. 1. An example graph and a possible spanning tree.

3.2 Maintaining Spanning Trees

A straightforward index-based solution for connectivity queries in dynamic graphs is to maintain a spanning tree for each connected component. Given a connected component C, a spanning tree is a connected subgraph of C including all vertices with the minimum number of edges. The subgraph is a tree structure clearly. An example graph and its spanning tree are presented in Figure 1. All tree edges are represented as solid lines, and all non-tree edges are represented as dashed lines. Given two query vertices, we can locate the root of each vertex by continuously scanning the tree parent. If the tree roots of two query vertices are the same, they are in the same spanning tree and the same connected component.

Maintaining simple spanning trees for all connected components is not challenging. For inserting a new edge (u, v), nothing needs to be done if two vertices have the same root. We call this case *non-tree edge insertion.* Otherwise, they are from different spanning trees, and we need to merge them. We call this case *tree edge insertion.* To merge two trees, we pick one vertex u of the inserted edge and rotate its spanning tree T_u to make u be the root. Then we can add T_u as a child subtree of v in its spanning tree T_v. For deleting an edge (u, v), nothing happens if (u, v) is a non-tree edge. Deleting a tree edge will immediately divide the spanning tree into two smaller trees. Then we need to identify if there is another edge connecting the two spanning trees. Once such an edge is found, we process it as the tree edge insertion.

3.3 The State-of-the-Art

Recently, Qing et al. [3] proposed a solution for connectivity query processing in dynamic graphs. Their solution is called D-Tree. They maintain a spanning tree with additional properties for each connected component to improve the average query efficiency. Their method is based on the following lemmas.

LEMMA 3.1. *The average costs of evaluating connectivity queries by spanning trees is optimal if each tree T minimizes $S_d(T)$, where $S_d(T)$ is the sum of distances between root and descendants in T, i.e., $S_d(T) = \sum_{u \in V(T)} depth(u)$. [3]*

Definition 3.2. (CENTROID) Given a spanning tree T, a centroid of T is a vertex with the smallest average distance to all other vertices.

LEMMA 3.3. *The average cost of each connectivity query by spanning trees is optimal if each tree is 1) rooted in the centroid, and 2) a BFS tree, i.e., the distance from every vertex to the root is minimal. [3]*

It is already very expensive to maintain just a valid BFS tree or a spanning tree rooted in the centroid. Therefore, D-Tree develops several strategies to reduce the average depth in tree maintenance. We summarize them into two categories.

Centroid Heuristic. The centroid heuristic rotates spanning trees for certain cases and aims to reduce the average depth without changing the tree edges. Specifically, when scanning vertices in tree updates, they attempt to locate the centroid of the spanning tree and rotate the tree so that the centroid is the root of the tree. To this end, they observe that given a spanning tree T rooted in its centroid c, the subtree size of every child of c in T is not larger than $|T|/2$. Therefore, once the subtree size of vertex u in T is larger than $|T|/2$ and all children of u are not, they identify that u is close to the centroid and make the tree rooted in the new root u.

BFS Heuristic. BFS heuristics reduce the average depth when updating the tree structure (i.e., changing tree edges), which happens in both edge insertion and edge deletion. When it is required to merge one tree T into the other tree T', the BFS heuristic pick a vertex u in T that has a non-tree neighbor v in T' with the lowest depth in T'. Then it rotates T to be rooted in u and adds the tree T as a subtree of v.

Algorithms of D-Tree. We describe the process of D-Tree below. In addition to maintaining the parent and the corresponding vertex id for each tree vertex like the spanning tree, D-Tree maintains the subtree size, children, and non-tree neighbors for each vertex. Subtree size is used for the centroid heuristic. Children and non-tree neighbors are used to efficiently find a replacement edge when deleting a tree edge. For query processing, as a by-product of continuously scanning parents for each vertex, they check if the visited child of the root has over half tree vertices in the subtree. If so, they apply the centroid heuristic and rotate the tree. Their query time complexity is $O(h)$ where h is the average depth.

For non-tree edge insertion, they check if the depth gap between two vertices u, v of the edge is larger than 1. If so, they pick one tree edge between u and v to cut the tree and apply the BFS heuristic to merge two trees by connecting u and v. For tree edge insertion, they pick and rotate a smaller tree and merge it into the bigger one. Note that every time a tree is updated, we need to update the subtree size for influenced vertices. Meanwhile, we rotate the tree once we meet a vertex satisfying the centroid property. They also need to maintain children and non-tree neighbors for each vertex even though they are not used in edge insertion. The time complexity of edge insertion time is $O(h \cdot \mathsf{nbr}_{\mathsf{update}})$, where $\mathsf{nbr}_{\mathsf{update}}$ is the time complexity to insert or delete an item from the children set and non-tree neighbors. If a hash set structure is used, $\mathsf{nbr}_{\mathsf{update}}$ can be reduced to a small constant but much memory will be used for a large number of buckets. If a balanced binary search tree is used, the running time is $O(h \cdot \log d)$ where d is the average degree.

The tree is immediately split into two trees when deleting a tree edge. They search non-tree neighbors of all vertices in the smaller tree to find a replacement edge to connect two trees. Given multiple replacement edge candidates, they apply the BFS heuristic to link the tree to the vertex with the lowest depth. The time complexity of edge deletion is $O(h \cdot \mathsf{nbr}_{\mathsf{scan}})$ if non-tree neighbors of each vertex can be scanned in linear time. Note that h is the average depth and is also the average subtree size of each vertex in the tree. The dominating cost is to scan non-tree neighbors for all vertices in the smaller tree. There are still some Euler tour-based existing solutions, such as HDT [8, 9]. However, as shown in our experimental results, D-Tree is more efficient than HDT. Therefore, we mainly introduce D-Tree here. We will also introduce some other related works in Section 8.

4 Revisiting D-Tree

D-Tree makes many efforts to reduce the average depth in the spanning tree. However, certain heuristics yield marginal benefits for the average depth but sacrifice much efficiency as a trade-off. To improve the overall efficiency, we follow the framework of D-Tree and propose an improved lightweight version called ID-Tree (Improved Dynamic Tree) in this section. We discuss our implementations for edge insertion and edge deletion in this section.

4.1 Motivation

Our solution is motivated by the following observations.

Observation 1. *Centroid heuristics in query processing of D-Tree may not help with improving query efficiency.*

As introduced in Section 3.3, D-Tree may rotate the tree in query processing to make the new root close to the centroid. Even though it just additionally brings constant time complexity, avoiding unnecessary tree updates may improve the query efficiency. One may suspect that avoiding this step will increase the average depth, which reduces the average query efficiency to some extent. We respond with the following lemma.

Lemma 4.1. *Given a spanning tree T rooted in u, let c be the centroid in T and T' be the tree with c as the root and the same tree edges as T. We have $maxdepth_T \leq 2 \cdot maxdepth_{T'}$ where $maxdepth_T$ represents the largest vertex depth in T.*

Proof. For any two vertices u and v in tree T' with the centroid c as the root, their distances to the centroid c must be less than $depth_{T'}(v)$. Therefore, there must exist a path passing through c between u and v, and the length of this path is less than $2 \cdot depth_{T'}(v)$. Therefore, the distance between any two vertices in $depth_T(v)$ must be less than $2 \cdot depth_{T'}(v)$. □

Lemma 4.1 shows that the theoretical max depth is still well-bounded even if we never rotate and balance the tree. On the other hand, D-Tree cannot guarantee to always maintain the centroid as the root. It is a trade-off between the depth and the additional efforts to reduce it. Note that the depth determines not only the query efficiency. Tree updates can also benefit from low depth since scanning from a vertex to the root happens in both edge insertion and deletion. Therefore, in our implementation, we still perform some rotation operations to make the root close to the centroid but never update the tree in query processing.

Improved Query Processing. Our query algorithm of ID-Tree is called ID-Query. The algorithm only searches the tree root of each query vertex and identifies if their roots are the same. Our performance studies show that the average depth of ID-Tree is competitive to that of D-Tree, and our improved query algorithm ID-Query is more efficient in most real datasets. We will further eliminate the dependency between the query efficiency and the average depth in Section 6, which achieves almost-constant query efficiency.

Observation 2. *Search replacement edges in processing tree edge deletion of D-Tree is expensive.*

Edge deletion in D-Tree takes much more time compared with edge insertion. It is caused by searching the replacement edge to reconnect trees when deleting a tree edge. Assume a tree edge (u, v) is deleted where v is the parent of u. D-Tree searches the neighbors of all vertices in the subtree rooted by u to find all edges that can reconnect the tree. Then they pick one of them based on certain heuristics. As a result, almost half edges in the graph will be scanned in the worst case. Our main target is to significantly improve the efficiency of edge deletion without sacrificing the efficiency of query processing and edge insertion. Our method can terminate searching the subtree once any replacement edge is found.

Observation 3. *Maintaining children and non-tree neighbors for each vertex is expensive.*

D-Tree maintains children and non-tree neighbors for finding the replacement edge as mentioned above. In exchange, the two sets require updating every time we rotate the tree or replace tree edges which frequently happens in query processing, edge insertion, and edge deletion. Maintaining them for each vertex is costly. Based on these observations, our implementation ID-Tree only maintains a subset of attributes in D-Tree for each vertex including:
- $parent(u)$: the parent of u in the tree;
- $st\_size(u)$: the size of subtree rooted in u.

4.2 Edge Insertion

Our improved algorithm for edge insertion is called ID-Insert. Before presenting details, we introduce three tree operators. They are the same as those in D-Tree except excluding updates for children and non-tree neighbors.

Algorithm 1: ID-Insert

```
Input: a new edge $(u, v)$ and the ID-Tree index
Output: the updated ID-Tree
1  $root_u \leftarrow$ compute the root of $u$;
2  $root_v \leftarrow$ compute the root of $v$;
   /* non-tree edge insertion */
3  if $root_u = root_v$ then
4      if $depth(u) < depth(v)$ then swap$(u, v)$;
5      if $depth(u) - depth(v) \leq 1$ then return;
       /* reduce tree deviation */
6      $w \leftarrow u$;
7      for $1 \leq i < \frac{depth(u)-depth(v)}{2}$ do
8          $w \leftarrow parent(w)$;
9      Unlink$(w)$;
10     Link(ReRoot$(u), v, root_v)$;
11     return;
   /* tree edge insertion */
12 if $st\_size(root_u) > st\_size(root_v)$ then
13     swap$(u, v)$;
14     swap$(root_u, root_v)$;
15 Link(ReRoot$(u), v, root_v)$;
```

- ReRoot(u) rotates the tree and makes u as the new root. It updates the parent-child relationship and the subtree size attribute from u to the original root. The time complexity of ReRoot() is $O(depth(u))$.
- Link$(u, v, root_v)$ adds a tree T_u rooted in u to the children of v. $root_v$ is the root of v. Given that the subtree size of v is changed, it updates the subtree size for each vertex from v to the root. We apply the centroid heuristic by recording the first vertex with a subtree size larger than $st\_size(root_v)/2$. If such a vertex is found, we reroot the tree, and the operator returns the new root. The time complexity of Link() is $O(depth(v))$.
- Unlink(u) disconnect the subtree of u from the original tree. All ancestors of u are scanned to update the subtree size. The time complexity of Unlink() is $O(depth(u))$.

We present the pseudocode of the improved edge insertion in Algorithm 1. Most processes are the same as that of D-Tree. We first conduct a connectivity query to identify if two vertices are in the same tree. Lines 3–11 apply the BFS heuristic for non-tree edge insertion. When the depth gap between u and v is over 1, a major difference from D-Tree is about the strategy of BFS heuristic in Line 7. D-Tree uses the threshold $depth(u) - depth(v) - 2$ instead of $\frac{depth(u)-depth(v)}{2}$. To reduce the average depth, we add half vertices from u to its ancestor with the same depth of v to the subtree rooted in u (Lines 7–9). Then we add the updated tree rooted in u to the children of v.

Example 4.2. Figure 2 shows the different strategies of BFS heuristic between D-Tree and our ID-Tree. We insert a non-tree edge (u, r). We have $depth(u) = 6$ and $depth(r) = 0$. Based on the strategy of D-Tree, we add u together with its three ancestors as a child subtree of r, which is presented in the middle figure. However, based on our strategy, we add u together with its two ancestors as a child subtree of r, which is presented in the right figure. The average depth of the tree is reduced from 1.9 to 1.7 in this example.

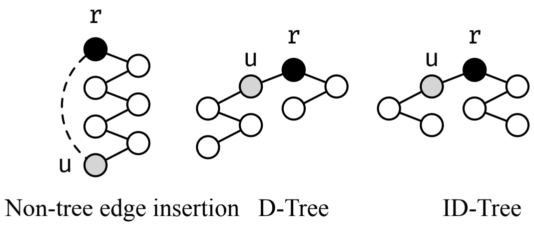

Fig. 2. Non-tree edge insertion in D-Tree and ID-Tree.

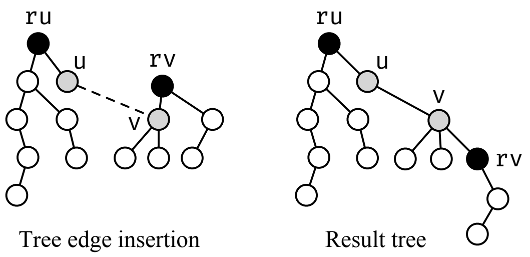

Fig. 3. An example of tree edge insertion for ID-Tree.

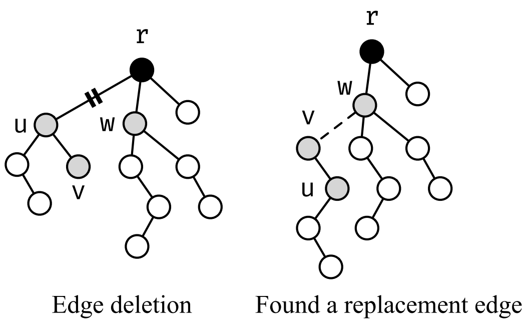

Fig. 4. An example of tree edge deletion for ID-Tree.

For tree edge insertion (Lines 12–14), our process is the same as D-Tree, which merges a smaller tree into a bigger tree.

Example 4.3. An example for tree edge insertion is shown in Figure 3. After inserting (u, v), we reroot the tree rooted by rv to v and add the updated tree rooted by v to the children of u.

4.3 Edge Deletion

Our improved edge deletion algorithm is called ID-Delete. The pseudocode is presented in Algorithm 2. Similar to D-Tree, we need to search for a replacement edge when a tree edge is deleted, and we always search from the smaller tree (Line 4). Compared with D-Tree, our strategy to find a replacement edge is completely different. The general idea is to immediately terminate the search once a replacement edge is found. We use a queue Q to maintain all visited vertices. In Line 6, S maintains the set of all visited vertices in the subtree of u. Without children and non-tree neighbors, we directly search graph neighbors (Line 9) for each vertex popped from the queue. Lines 11–13 are for children. We add them to the queue for further exploration. For each non-tree neighbor y (Lines 14–24), we check if y is from the different tree by scanning the ancestors of y (Lines 16–21). An optimization here is to terminate the current iteration once we reach a vertex recorded in the subtree of u (Lines 17–19). If $succ$ keeps positive after all iterations of Lines 16–21, it means we already reach the root of y, and y belongs to a different tree. As a result, we link two trees via the edge (x, y) (Lines 22–24). Otherwise, all visited vertices in Line 17 are in the subtree of u, and we

Algorithm 2: ID-Delete

```
**Input:** an existing edge $(u, v)$ and the ID-Tree
**Output:** the updated ID-Tree
1  if $parent(u) \neq v \wedge parent(v) \neq u$ then return;
2  if $parent(v) = u$ then swap$(u, v)$;
3  $root_v \leftarrow$ Unlink$(u)$;
   /* reduce the worst-case time complexity of searching replacement edge in
      subtree                                                               */
4  if $st\_size(root_v) < st\_size(u)$ then swap$(u, root_v)$;
   /* search subtree rooted in $u$                                          */
5  $Q \leftarrow$ an empty queue, $Q.push(u)$;
6  $S \leftarrow \{u\}$;
   /* $S$ maintains all visited vertices                                    */
7  while $Q \neq \emptyset$ do
8      $x \leftarrow Q.pop()$;
9      foreach $y \in N(x)$ do
10         if $y = parent(x)$ then continue;
11         else if $x = parent(y)$ then
12             $Q.push(y)$;
13             $S \leftarrow S \cup \{y\}$;
14         else
15             $succ \leftarrow$ true;
16             foreach $w$ from $y$ to the root do
17                 if $w \in S$ then
18                     $succ \leftarrow$ false;
19                     break;
20                 else
21                     $S \leftarrow S \cup \{w\}$;
22             if $succ$ then
23                 $root_v \leftarrow$ Link(ReRoot$(x)$, $y$, $root_v$);
24                 return;
```

continue to search for the next possible replacement edge. Maintaining all visited vertices in S guarantees each vertex is scanned only once for the whole process of searching for replacement edges.

Example 4.4. A running example of tree edge deletion is shown in Figure 4. In the left figure, (r, u) is deleted, and the tree is split into two subtrees. We search for the replacement edge from the smaller tree, which is rooted in u. Assume that a replacement edge (v, w) is found. We reroot the smaller tree to v and link two trees by connecting (v, w). The right figure shows the final result.

5 Theoretical Analysis

5.1 Edge Insertion

THEOREM 5.1. *The time complexity of Algorithm 1 is $O(h)$, where h is the average depth.*

Proof. Computing the roots of u and v requires $O(h)$ time. For non-tree edge insertion, if the depth gap between u and v is less than 1, no other operation is needed; otherwise, an existing edge needs to be unlinked and a new inserted edge needs to be linked, which also takes $O(h)$ time. For tree edge insertion, the two trees can be linked directly after rerooting, which also takes $O(h)$ time. □

5.2 Edge Deletion

Our strategy to find a replacement edge is simple but efficient in both theory and practice. Even though we scan graph neighbors for a vertex from the queue, we show that the search space is quite small by the following lemma. We assume that the distribution of vertices and edges is uniform.

Lemma 5.2. *The expected total number of iterations of Line 9 in Algorithm 2 is $O(h)$, where h is the average tree depth.*

Proof. The sum of the subtree size of all vertices in a tree is equal to the sum of the depth of all vertices. Therefore, the average subtree size of a vertex can be considered as $O(h)$. In Algorithm 2, we need to traverse tree edges and non-tree edges in the subtree. The tree edges are visited at most $O(h)$ times. For non-tree edges, since the BFS search is always performed on the tree with the smaller *st_size* rooted at u, each non-tree edge has a probability of being the replacement edge greater than 1/2. Therefore, the expected total number of iterations of Line 9 is still $O(h)$. □

Based on Lemma 5.2, we show the time complexity for our improved algorithm for edge deletion.

Theorem 5.3. *The expected time complexity of Algorithm 2 is $O(h)$.*

Proof. For each iteration of Line 16 in Algorithm 2, we traverse the path from y to the root, which takes $O(h)$ time. Based on Lemma 5.2, the expected number of visits to non-tree edges is bounded by 2. Therefore, the expected time complexity of Algorithm 2 is $O(h)$. □

6 constant-time Query Processing

The ideas of D-Tree and our improved version in Section 4 are to maintain a near balanced spanning tree since the query time depends on the depth of each vertex. In this section, we eliminate the dependency and **further improve the query efficiency to almost-constant**. Meanwhile, we **bound the same theoretical running time for edge insertion and edge deletion**.

6.1 Utilizing the Disjoint-Set Structure

Our idea is inspired by the disjoint-set data structure [25]. It provides two operators, Find and Union, which are presented in Algorithm 3. Find returns an id for the set containing the input item, and Union merges the sets of two given items. The structure maintains each set as a tree and uses the tree root to represent the set. Two crucial optimizations are adopted. The first is path compression. When path compression is applied, in the path from a query item to the root in Find, the algorithm connects every visited item to the root directly. The second operation is union by size. Given that the *size* of each item in Union represents the number of items in the tree, we always merge the smaller tree with the larger tree. With these two optimizations, the amortized time complexity for both Find and Union is $O(\alpha(n))$, where n is the number of items, and $\alpha()$ is the inverse Ackermann function. Due to the extremely slow growth rate of $\alpha()$, it remains less than 5 for all possible values of n that can be represented in the physical universe.

Using disjoint set is a fundamental method to detect all connected components and is also natural to deal with scenarios with only edge insertions. However, given an edge deletion disconnecting two connected components, it is hard to split a set based on the disjoint-set structure. Identifying all vertices belonging to one of the resulting connected components is challenging. We define DS-Tree as the tree structure maintained in Algorithm 3. Note that some additional attributes will be maintained for each vertex in DS-Tree, which are used for edge deletion and will be introduced later.

Algorithm 3: Disjoint-set operators

```
Procedure Find($x$)
    if $x.parent \neq x$ then
        $x.parent \leftarrow$ Find($x$);
        return $x.parent$;
    return $x$;
Procedure Union($x, y$)
    $x \leftarrow$ Find($x$);
    $y \leftarrow$ Find($y$);
    if $x = y$ then return;
    if $x.size > y.size$ then swap($x, y$);
    $x.parent \leftarrow y$;
    $y.size \leftarrow x.size + y.size$;
```

Our idea is to simultaneously maintain a ID-Tree and a DS-Tree for each connected component. We utilize DS-Tree to improve the performance of query processing and edge insertion. We update DS-Tree with the help of ID-Tree for edge deletion.

We introduce the data structure of DS-Tree in Section 6.2. In Section 6.3, we study the details of several DS-Tree operations which serve in our final algorithms for edge insertions and deletions. Details of the final query processing algorithm will also be covered in Section 6.3. Then, we introduce our final edge insertion algorithm and edge deletion algorithm in Section 6.4 and Section 6.5, respectively. The complexity analysis of these two algorithms is also covered there.

6.2 DS-Tree Structure

Optimal Children Maintenance in DS-Tree. We discuss the data structure of DS-Tree in this subsection. We start by considering the attributes of each vertex in DS-Tree. For an edge deletion, it may require removing several vertices from DS-Tree, and we need to connect all children for each removed vertex back to the tree. Unlike D-Tree, tree edges in DS-Tree may not be graph edges, which makes searching children from scratch (like Line 9 of Algorithm 2) not feasible. Therefore, a data structure to maintain children of each vertex in DS-Tree is expected to support the following tasks.

- Task 1. Deleting a vertex u from children of v;
- Task 2. Scanning all children of u;
- Task 3. Inserting a vertex u into the children set of v.

The first two tasks are required when we remove u from a DS-Tree. The last task is required when we delete the parent of u or take the union of two DS-Trees rooted in u and v, respectively. A straightforward idea is to use a hash table. However, even if a large number of buckets are used to achieve a high efficiency for insertion and deletion, scanning all children in the second task is time-consuming. To overcome this challenge, we adopt a doubly-linked list (DLL) structure to maintain all children for each vertex in DS-Tree. Specifically, we use a structure called DSnode to represent each vertex in DS-Tree and maintain the following attributes for DSnode.

Note that we do not maintain the subtree size for each vertex. This is because we only use the subtree size of the DS-Tree root when linking two DS-Trees. The subtree size of a vertex u in DS-Tree is the same as that in ID-Tree if u is the root in both trees. The children attribute points to the first child in DLL. To remove a child u, we connect the previous child (DSnode.*pre*) and the

- *id*	// id of the corresponding vertex;
- *parent*	// pointer to the parent's DSnode in *DS*-tree;
- *pre*	// previous pointer in the DLL of the parent's children;
- *next*	// next pointer in the DLL of the parent's children;
- *children*	// start position of the DDL of children.

next child (DSnode.*next*) in DLL. To insert a child u, we add u to the beginning of the DLL. The following lemma holds.

Lemma 6.1. *Inserting a new child or deleting an existing child for a vertex in DS-Tree is completed in $O(1)$ time.*

For task 2, it is clear to see that all children can be scanned following the DLL, and the time only depends on the number of children, which is also optimal.

6.3 DS-Tree Operators

We introduce several operators to manipulate DS-Trees which are used in our final algorithms. We add a suffix DS to distinguish certain operators from those of D-Tree. All operators are presented in Algorithm 4, and we will show that all of them can be implemented in amortized constant time, which is optimal.

UnlinkDS and LinkDS. We start with operators that are straightforward to be implemented. Changing the parent of a tree node often happens in manipulating DS-Trees, and a typical scenario is the path compression optimization to find the root in a disjoint-set tree (i.e., all visited vertices are assigned as the children of the root). UnlinkDS disconnects the vertex u from its parent. It removes u from the children DLL of its parent. For ease of presentation, we add virtual vertices at the beginning and end of DLL respectively. In this way, the *pre* pointer (Line 2) and the *next* pointer (Line 3) are not Null.

LinkDS adds a vertex to the children DLL of the other vertex and is a simplified version of Union. In our final update algorithms, we identify the root and the size of each DS-Tree before invoking LinkDS. Therefore, unlike the original Union operator, we do not need to execute Find to find the root of each vertex and compare the size of two trees. We set v as the parent of u in Line 8 and add u to the children DLL of v. For ease of presentation, we add virtual vertices at the beginning and end of DLL respectively. DSnode(v).*children* (Line 9) points to the virtual beginner of children DLL of v. In this way, the *pre* pointer (Line 11) and the *next* pointer (Line 12) are not Null. The strategy of LinkDS is consistent with the Union operator of disjoint-set structure [25]. The time complexity of both UnlinkDS and LinkDS is $O(1)$.

FindDS for Query Processing. Based on UnlinkDS and LinkDS, we reorganize the Find operator in the original disjoint set based on the attributes of DSnode. The operator is executed to find the root of each vertex. Recall that in our query processing algorithm for ID-Tree, nothing needs to be maintained. Therefore, we can process queries based on DS-Tree and do nothing for D-Tree. Our final query processing algorithm is called DND-Query. It invokes FindDS to find the root of each vertex and identifies if they are the same. The theoretical querying time is summarized below.

Theorem 6.2. *The amortized time complexity of* DND-Query *for connectivity query processing is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function of the number of vertices n in the graph.*

Proof. The strategy of DND-Query is consistent with the Find operator of Disjoint-Set structure [25]. DND-Query has the same time complexity as Find. □

Given that $\alpha(n) < 5$ as mentioned earlier, this makes an amortized constant time for our query processing algorithm in practice.

Isolate. The Isolate operator deletes a vertex from a DS-Tree. A DS-Tree may require to be split into two trees for edge deletion. To this end, we Isolate all vertices belonging to the smaller ID-Tree

Algorithm 4: DS-Tree operators

```
Procedure UnlinkDS($u$)
    DSnode($u$).$pre.next$ $\leftarrow$ DSnode($u$).$next$;
    DSnode($u$).$next.pre$ $\leftarrow$ DSnode($u$).$pre$;
    DSnode($u$).$parent$ $\leftarrow$ DSnode($u$);
    DSnode($u$).$pre$ $\leftarrow$ Null;
    DSnode($u$).$next$ $\leftarrow$ Null;
Procedure LinkDS($u, v$)
    /* union without find and comparing size */
    /* the input satisfies $st\_size(u) \leq st\_size(v)$ */
    /* union two DS-Trees */
    DSnode($u$).$parent$ $\leftarrow$ DSnode($v$);
    /* add $u$ to the new DLL */
    DSnode($u$).$pre$ $\leftarrow$ DSnode($v$).$children$;
    DSnode($u$).$next$ $\leftarrow$ DSnode($v$).$children.next$;
    DSnode($u$).$pre.next$ $\leftarrow$ DSnode($u$);
    DSnode($u$).$next.pre$ $\leftarrow$ DSnode($u$);
Procedure FindDS($u$)
    if DSnode($u$).$parent$ $\neq$ DSnode($u$) then
        $root$ $\leftarrow$ FindDS(DSnode($u$).$parent.id$);
        UnlinkDS($u$);
        LinkDS($u, root$);
        return $root$;
    return $u$;
Procedure Isolate($u$)
    /* assign children of $u$ to the root */
    $root_u$ $\leftarrow$ FindDS($u$);
    UnlinkDS($u$);
    foreach child $w$ of $u$ in DS-Tree do
        UnlinkDS($w$);
        LinkDS($w, root_u$);
Procedure ReRootDS($u$)
    $root_u$ $\leftarrow$ FindDS($u$);
    swap(DSnode($u$), DSnode($root_u$));
    DSnode($u$).$id$ $\leftarrow$ $u$;
    DSnode($root_u$).$id$ $\leftarrow$ $root_u$;
```

from the old DS-Tree and union all vertices which are isolated as a new DS-Tree. Instead of adding all children of u to its parent in the pseudocode, we first do a path compression and find the root of the DS-Tree. Then, we link each child to the root by executing the LinkDS operator (Line 25).

Lemma 6.3. *The amortized time complexity of* Isolate *is* $O(1)$.

Proof. The lemma is straightforward since the number of all children in the tree is n, and the amortized children number of each vertex is $O(1)$. □

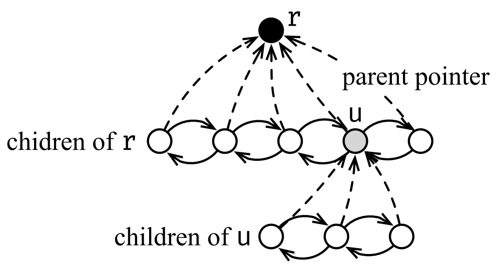

Fig. 5. An example of Isolate(u).

The Isolate operation removes a given vertex u from its connected component. In the operation, all children of u are added back to the DS-Tree as the children of the root, which guarantees the children are not removed. Note that given the path compression optimization, the children number of the DS-Tree root is very large, and that for all other tree vertices is very small. **Our final algorithm for edge deletion will guarantee to never execute** Isolate **for the root.**

Example 6.4. An example of executing Isolate is shown in Figure 5. Assume that r is the DS-Tree root. To delete the vertex u, we first connect the previous vertex and the next vertex in the doubly linked list. Then we connect the children of u to the root r.

RerootDS. The ReRootDS operator is crucial to keep the DS-Tree root consistent with that of ID-Tree. Several potential ways exist to reroot DS-Tree. One way is to rotate the DS-Tree similar to the ReRoot operator in ID-Tree where all tree edges do not change. Given the new root u, another way is to first delete u from the tree by invoking the Isolate operator and then assign u as the parent of the original root by invoking the LinkDS operator. However, both of the above methods increase the depth of vertices in DS-Tree, which reduces the efficiency of FindDS and may even break the amortized constant time complexity of FindDS.

Motivated by this, our method to reroot DS-Tree is to replace the new root u with the original root $root_u$ directly. By replacement, we mean to exchange the children and the parent of u and $root_u$. Note that the parent attribute of every child of $root_u$ will be pointed to u. Even if the amortized number of children for each tree vertex is $O(1)$ as proved in Lemma 6.3, the size of all children of $root_u$ can be very large. To improve the reroot efficiency, we store a DSnode pointer instead of a vertex id in the parent attribute for each DSnode. In this way, as shown in Lines 28–30 of Algorithm 4, we swap the corresponding DSnode objects of two vertices, and update their id for the new vertex. The time complexity of ReRootDS is $O(1)$.

6.4 The Final Edge Insertion Algorithm

For edge insertion, we Union two DS-Trees if a tree edge is inserted to connect two connected components. Additionally, we propose an optimization to improve the performance of both insertion and deletion. The optimization keeps the root of DS-Tree the same as that of ID-Tree. It benefits the following operations in edge insertion and deletion, respectively.

- *(a) Avoid searching ID-Tree root.* As shown in Algorithm 1, we need to search the root of ID-Tree frequently. Given that finding the ID-Tree root takes amortized constant time, keeping their roots consistent speeds up the edge insertion significantly.
- *(b) Avoid deleting ID-Tree root.* In Line 3 of Algorithm 2, we split the subtree T_u of u from the original tree if a replacement edge is not found. In our final edge deletion framework discussed later, we will delete all vertices in T_u from DS-Tree. Keeping the two roots of ID-Tree and DS-Tree consistent guarantees that the DS-Tree root is never deleted. An immediate challenge

```
Algorithm 5: DND-Insert
  Input: an existing edge $(u, v)$ and the DND-Trees index
  Output: the updated DND-Trees
1 $root_u \leftarrow$ FindDS($u$);
2 $root_v \leftarrow$ FindDS($v$);
3 Lines 3–14 of Algorithm 1;
4 ReRoot($u$);
5 LinkDS($root_u, root_v$);
6 Link($u, v, root_v$);
```

```
Algorithm 6: DND-Delete
  Input: an existing edge $(u, v)$ and the DND-Trees index
  Output: the updated DND-Trees
1 $(u, root_v, succ, S) \leftarrow$ ID-Delete($u, v$);
2 ReRootDS($root_v$);
3 if $succ$ then return;
  /* $u$ is the root of the smaller ID-Tree */
4 Isolate($u$);
5 $S \leftarrow S \setminus \{u\}$;
6 foreach $w \in S$ do
7     Isolate($w$);
8     LinkDS($w, u$);
```

after deleting the DS-Tree root is to identify a new root. Then we need to link all children of u back to the new root. Due to the path compression optimization, the number of children is extremely large. Connecting all children of the DS-Tree root to the new root is costly.

We present the final algorithm for edge insertion in Algorithm 5. The final index is called DND-Trees (short for ID-Tree and DS-Tree), and the insertion algorithm is called DND-Insert. In Lines 1–2, we find the root of each vertex in DS-Tree by invoking FindDS. Given that roots of two tree are consistent, fining root in DS-Tree is much more efficient. For non-tree edge insertion, we reorganize the ID-Tree same as Algorithm 1. Given that the tree root does not change, we do nothing for DS-Tree. For tree edge insertion, we need Union two DS-Trees given that two connected components are connected. After Line 14 of Algorithm 1, we already know $root_u$ and $root_v$ are roots of two original DS-Trees and the subtree size of $root_u$ is smaller than that of $root_v$. Note that in Line 4, we ReRoot the ID-Tree of $root_u$ to u but do not keep the root of DS-Tree consistent. This would not break the correctness since the tree will be merged into the larger tree, and the root of the final DS-Tree will be correct. In Line 5, we invoke LinkDS to assign $root_u$ as the child of $root_v$ in DS-Trees which essentially merges two DS-Trees. Finally, we link ID-Trees as before. Given the implementation details of all DS-Tree operators, we have the following theorem.

Theorem 6.5. *Algorithm 5 has the same running time complexity as Algorithm 1.*

6.5 The Final Edge Deletion Algorithm

Recall that in Algorithm 2 for edge deletion, we first unlink the ID-Tree. After Line 4 of Algorithm 2, u is the root of the smaller tree for searching replacement edge, and $root_v$ is the root of the larger tree. To update the corresponding DS-Tree, we still consider two cases based on the existence of the replacement edge. If a replacement edge is found, the connected component does not update,

Dataset	name	n	m	Type	$h_{ID-Tree}$	h_{D-Tree}	$\|S\|$	$\#search$	$maxd$
dynamic-dewiki[1]	DE	2,166,670	86,337,879	Temporal, Hyperlink	2.828	2.778	1.015	1.001	4
stackoverflow[2]	ST	2,601,978	63,497,050	Temporal, QA	2.509	2.445	1.037	1.005	5
soc-bitcoin[3]	BI	24,575,383	122,948,162	Temporal, Transaction	5.384	6.350	1.133	1.477	8
soc-flickr-growth[3]	FL	2,302,926	33,140,017	Temporal, Social	1.163	1.190	1.424	1.055	6
edit-enwiki[1]	EN	50,757,444	572,591,272	Temporal, Edit	2.472	2.160	1.008	1.002	6
delicious-ti[1]	TI	38,289,742	301,183,605	Temporal, Feature	2.600	2.592	1.051	1.029	6
delicious-ui[1]	UI	34,611,304	301,186,579	Temporal, Interaction	3.550	3.321	1.012	1.018	5
yahoo-song[1]	YA	1,625,953	256,804,235	Temporal, Rating	2.197	2.554	1.000	1.000	3
LiveJournal[1]	LI	4,846,610	42,851,237	Unlabeled, Social	3.911	4.272	1.068	1.002	3
twitter_mpi[1]	TM	52,579,683	1,614,106,187	Unlabeled, Social	2.353	2.462	1.015	1.000	2
twitter-2010[1]	T2	41,652,230	1,202,513,046	Unlabeled, Social	2.334	3.470	1.013	1.000	2
friendster[1]	FR	124,836,180	1,806,067,135	Unlabeled, Social	1.855	2.640	1.047	1.003	2
uk-2007[4]	UK	133,633,040	4,663,392,591	Unlabeled, Hyperlink	4.003	5.249	1.818	1.002	3
CTR[5]	CT	14,081,817	16,933,413	Unlabeled, Road	2,356.592	1,883.230	3.279	6.541	3
W[5]	W	6,262,105	7,559,642	Unlabeled, Road	1,410.453	1,404.410	4.194	10.840	3
road-usa[3]	US	23,947,348	28,854,312	Unlabeled, Road	2,864.871	2,726.730	3.202	6.791	3

Table 2. The Description of Dataset. h is the average vertex depth in different spanning trees. m is the number of edges and n is the number of vertices. $|S|$ is the average size of S in Algorithm 6. $\#search$ is the actual average number of iterations which is mentioned in Lemma 5.2. $maxd$ is the maximum depth of vertex on DS-tree during query phase.

and the tree root is $root_v$ in Line 23 of Algorithm 2. We only reroot the DS-Tree to $root_v$ in this case. If a replacement edge is not found, we need to split the original DS-Tree into two trees for two resulting ID-Trees. To this end, our idea is to delete every vertex belonging to the subtree of u in ID-Tree from the DS-Tree. Then, we Union all deleted vertices and form the new DS-Tree. An immediate challenge is how to identify all vertices belonging to the subtree of u in ID-Tree. We discuss this in the following lemma.

Lemma 6.6. *Given u in Line 5 of Algorithm 2, if no replacement edge is found (i.e., succ turns false), S is the set of all vertices in the subtree rooted in u when the algorithm terminates.*

Based on Lemma 6.6, we present our final deletion algorithm, called DND-Delete, for DND-Trees index in Algorithm 6. After updating ID-Trees, we first reroot the DS-Tree to $root_v$ in Line 2. Given $succ$ = true in Line 3, we terminate the algorithm since the connected component does not update. Otherwise, we delete all vertices in S from the DS-Tree in Lines 4–8. We first delete u since u will be the root of the new DS-Tree. Then, we iteratively delete each vertex w from S and link w to u. S contains all vertices in the subtree of u in the ID-tree. Given that the parent of u is $root_v$ (Line 1), $root_v$ is not in S. In Line 2 of Algorithm 6, we update the root of the disjoint set tree to $root_v$. Therefore, we never Isolate the root of the disjoint-set tree. From the perspective of disjoint set, we perform the Union operation on u and w in Line 8. As introduced in Section 6.4, LinkDS(w, u) directly assigns w as a child of u since the subtree size of u is guaranteed larger than that of w in DS-Tree and both of them are roots before Union.

Theorem 6.7. *Algorithm 6 has the same expected running time complexity as Algorithm 2.*

Proof. On the basis of Algorithm 2, Algorithm 6 also needs to traverse all the vertices in S to decompose the DS-tree. Since all DS-Tree operations are $O(1)$, the added operation is bounded by $|S|$. Algorithm 2 is also bounded by $|S|$, so the complexity of the two deletion operations is still the same. □

7 Performance Studies

Setup. All algorithms are implemented in C++ and compiled with O3 level optimization. The experiments are conducted on a single machine with Intel Xeon Gold 6248 2.5GHz and 768GB RAM. All results are averaged over ten runs on the same machine.

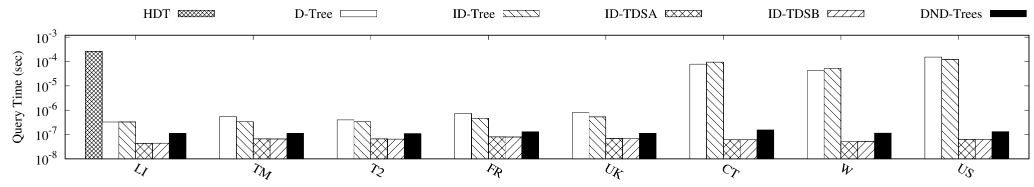

Fig. 6. Query time of unlabeled graphs.

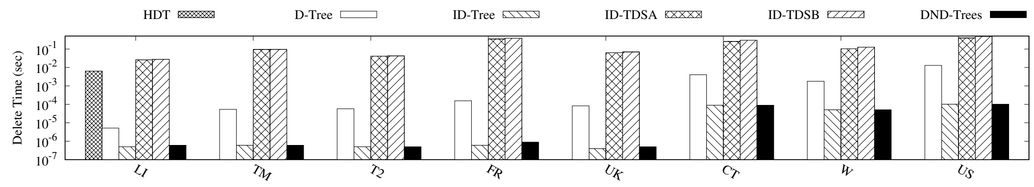

Fig. 7. Delete time of unlabeled graphs.

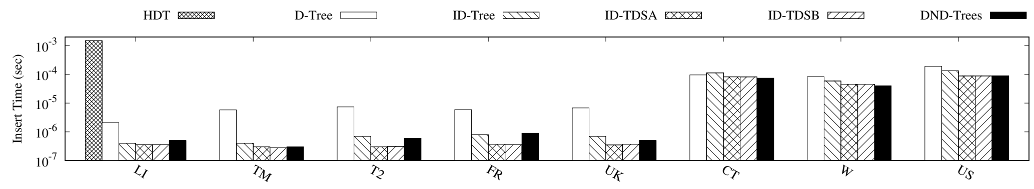

Fig. 8. Insert time of unlabeled graphs.

Dataset. We evaluate sixteen real datasets from different domains (Table 2). These datasets can be found at konect[1], Stanford Large Network data set Collection[2], Network Repository[3], LAW[4] and DIMACS[5]. Eight out of sixteen datasets are temporal graphs, and the rest are unlabeled graphs. Our algorithms can be used for recommendation in social networks [16], transaction analysis in trading networks [14], path planning in road networks [5], etc.

Competitors. We evaluate the performance of connectivity queries and update operations for the following methods:

- **DND-Trees.** Our final algorithm includes all optimizations.
- **ID-Tree.** Our algorithm shown in section 4.
- **D-Tree.** The algorithm proposed by Chen et al. [3]. The source code from [3] is in Python. We re-implement it in C++.
- **ID-TDSA.** A baseline to combine ID-Tree and the disjoint set. When a tree edge is deleted in ID-Tree and no replacement edge exists, we reconstruct the whole disjoint set.
- **ID-TDSB.** The other baseline to combine ID-Tree and the disjoint set. When a tree edge is deleted in ID-Tree and no replacement edge exists, we reconstruct the disjoint set for all vertices in the old connected component containing the deleted edge.
- **HDT.** The algorithm proposed by Holm et al. [8, 9]. The code is from an experimental paper [11].

7.1 Performance in Unlabeled Graphs

For a general unlabeled graph, we first build an index with all the edges of the entire dataset. Then we randomly delete 100,000 edges and then insert those 100,000 edges back into the graph. We calculate the average running time for insertions and deletions, respectively. For query efficiency,

[1]http://konect.cc/networks/
[2]https://snap.stanford.edu/data/
[3]https://networkrepository.com/
[4]http://law.di.unimi.it/datasets.php
[5]http://www.diag.uniroma1.it/challenge9/download.shtml

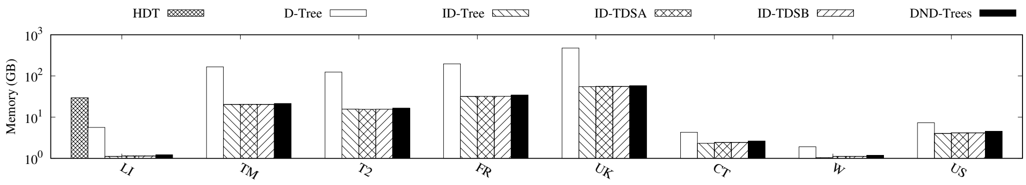

Fig. 9. Memory of unlabeled graphs.

we randomly generate 50, 000, 000 vertex pairs and identify whether they are connected. Due to the large average vertex depth of the road network, queries are not efficient. For them we query 100, 000 times. We calculate the average time of a query operation. For any of the test cases in our paper, DND-Trees can complete the entire operation in less than 1, 000 seconds. We do not report the results of tests that take longer than 12 hours. A case executes for more than 12h is mostly due to its very low deletion efficiency, indicating that the algorithm is not suitable for processing large-scale data. All subsequent experiments also follow this setting.

Query Processing. Figure 6 shows the average query time in 11 unlabeled graphs. The query efficiency of DND-Trees is much higher than that of ID-Tree and D-Tree, and very close to ID-TDSA and ID-TDSB. For the instance of US, DND-Trees is two orders of magnitude faster. The efficiency of both ID-Tree and D-Tree is related to the average depth. Their query time is different but the overall query efficiency of ID-Tree and D-Tree is similar.

Deletion. Figure 7 shows the average time of the edge deletion. On all datasets, the deletion efficiency of DND-Trees is significantly higher than that of D-Tree. Two orders of magnitude speedup is achieved on TM, UK and US. We can also see that the additional cost of DND-Trees beyond ID-Tree is marginal, which supports Theorem 6.5 and Theorem 6.7. ID-TDSA and ID-TDSB are much slower than DND-Trees, because they need to visit nearly all the tree edges in ID-Tree.

Insertion. Figure 8 shows the average time of edge insertion. Our final algorithm is much faster than D-Tree. Note that in certain small datasets (LI and FR), DND-Trees is a little slower than ID-Tree. This is because when the average depth of the ID-Tree itself is relatively small, although the DS-Tree speeds up the root-finding operation, additional time is required to maintain the DS-Tree.

Memory. Figure 9 shows the memory usage. Compared to D-Tree, ID-Tree does not need to maintain children and non-tree neighbors and its memory is smaller. Compared to ID-Tree, DND-Trees include DS-Tree structure and need a little more memory.

Different types of update operations. Figure 10 shows the efficiency of different types of update operations (inserting tree edges, inserting non-tree edges, deleting tree edges and deleting non-tree edges). We first initialize the spanning tree based on all edges in the dataset and obtain the set of tree edges and the set of non-tree edges. Next, we delete all non-tree edges and calculate the efficiency of non-tree edge deletion. Then, all tree edges are deleted, and the efficiency of tree edge deletion is calculated. Next, we insert all tree edges and calculate the efficiency of tree edge insertion. Finally, we insert all non-tree edges and calculate the efficiency of non-tree edge insertion. We can find that the update efficiency of tree edges is faster than that of non-tree edges, and the gap increases when h increases.

7.2 Performance in Sliding Windows

We investigate three algorithms over different sizes of sliding windows in temporal graphs. For each temporal graph, we first compute the time span for the dataset. Then we vary the window size in 5%, 10%, 20%, 40% and 80% of its time span. We insert all edges in chronological order. When the time difference between the inserted new edge and the oldest edge in the window is greater than the time window size, the old edge is deleted. We record the average time of a sliding operation (inserting a new edge and deleting an expired edge). We also randomly query 50,000,000 times in

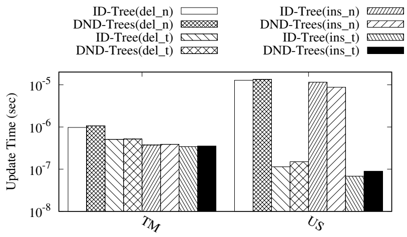

Fig. 10. Update time of tree and non-tree edges.

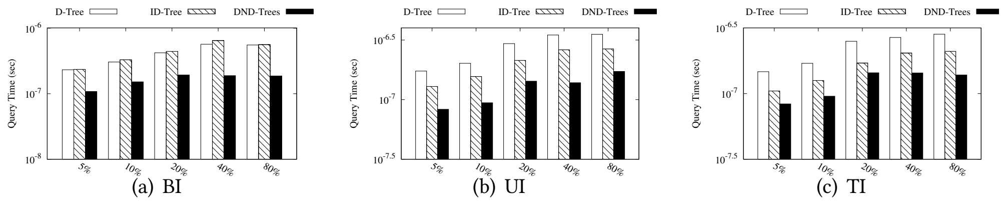

Fig. 11. Query time (vary window size).

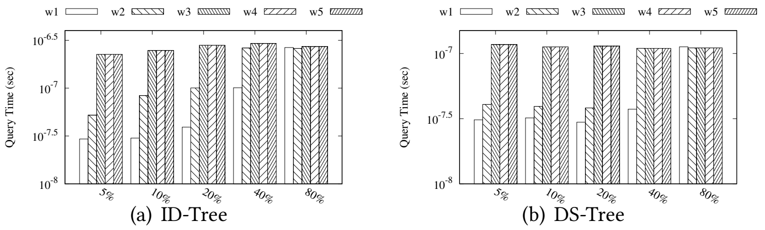

Fig. 12. Query time in different windows.

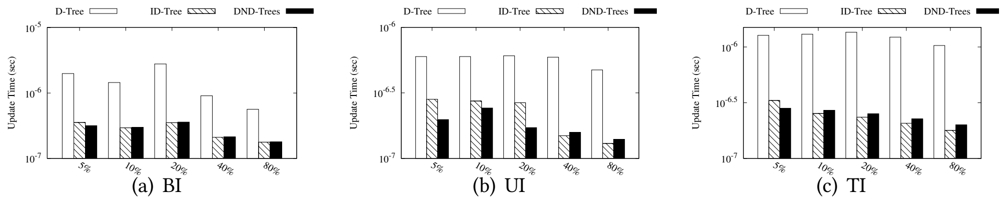

Fig. 13. Update time (vary window size).

the last time window and calculate the average query time. We report the representative datasets given the space limitation.

Query processing by varying window size. Figure 11 shows the average query time of different window sizes. As the time window size increases, the query time of both D-Tree and ID-Tree increases. However, the query time of DND-Trees is stable given its constant query time. Figure 12 shows the average query time in five different sliding windows. Due to space constraint, we chose dataset BI as an example. We select 5 windows evenly for querying during the sliding process, depending on the window size. Specifically, when the window size is 40%, we select the following five windows: 0%–40%, 15%–55%, 30%–70%, 45%–85% and 60%–100%. Some of the earlier windows have fewer temporal edges, and their queries are more efficient, as influenced by the distribution of temporal edges. However, their query efficiency stabilizes when there are enough temporal edges in the window.

Updating by varying window size. Figure 13 shows the average update time of different window sizes. The DND-Trees and ID-Tree are much faster than D-Tree. As the window size increases, the time variation of all three algorithms is not obvious. For the sliding window updates, deleting an edge is the dominating cost because of searching the replacement edge. As the window size increases, the number of edges increases, and the graph becomes denser. As a result, the probability of deleting a non-tree edge is higher. Compared with the case of deleting a tree edge, deleting a non-tree edge is much more efficient.

7.3 Average depth of spanning tree

The efficiency of all algorithms is related to the average tree depth h except for our final query algorithm. $h_{ID-Tree}$ and h_{D-Tree} in Table 2 show the average tree depths in different graphs. *maxd* in Table 2 shows the maximum depth of vertex on DS-Tree during query phase. For temporal datasets, they refer to the average depth where the time window is set to 40%. Even though we relax certain heuristics to reduce average tree depth, the result shows that our tree depth is still competitive. Furthermore, experimental results show that *maxd* is very small, and the depth of the DS-Tree can be considered as a constant.

7.4 Experimental Analysis of Delete Operation

Table 2 reports the average value of $|S|$ in Algorithm 6 when *succ* is *true* in Line 3. For temporal datasets, h refers to the case where the time window is set to 40% of the time span. Compared to ID-Tree, the additional cost of DND-Trees appears when no replacement edge can be found. We disconnect the DS-Tree as shown in Algorithm 6. The cost depends on the size of S. From Table 2, $|S|$ is very small which supports our theoretical results and proves that the additional cost is negligible. We also report the actual average number of iterations of Line 9 in Algorithm 2. As shown in Table 2, *#search* is small and it is in line with Lemma 5.2. This demonstrates the high efficiency of our edge deletion operation.

8 Related Work

Connectivity in Undirected Graphs. Initially, connectivity algorithms were designed for updating spanning trees either for edge insertions [24] or for edge deletions [22]. Henzinger and King proposed [6, 7] a method of representing spanning trees through Euler tours [26]. It adds information to enable early termination of the search for a replacement edge. Holm et al. [8, 9] proposed a new structure that makes the update efficiency of the index reach $O(\log^2 n)$. Huang et al. [10] further theoretically reduced the time complexity to $O(\log n(\log\log n)^2)$. Chen et al. [3] introduce a new data structure, called the dynamic tree (D-Tree). A detailed analysis of D-Tree can be found in Section 3.3. Connectivity maintenance in streaming graphs is studied in [23].
Connectivity/reachability in Directed Graphs. Much of the prior research on reachability of directed graphs [2, 4, 12, 28, 33] focuses on labeling schemes. Those approaches are typically not suited for undirected graphs. They can be classified into two main categories: interval labeling and 2-HOP labeling [13]. Those methods can also be modified for dynamic graphs. Optimal Tree Cover (Opt-TC) [1] is based on interval labeling. It is one of the initial works to tackle the incremental maintenance of the index in dynamic graphs. Based on 2-HOP labeling, some incremental maintenance methods are proposed in [2, 20, 21]. However, they do not support efficient delete operations. A recent data structure for labeling, known as DBL [13], does support undirected and directed graphs. It only allows edge insertions in graphs, and the construction of DBL is time-consuming due to the need of performing a BFS on the corresponding connected components.
Other types of graphs. There are many related studies on other types of graphs related to connectivity queries. In temporal graphs, the span-reachability query aims to answer the reachability

of any time window [29, 30]. Qiao et al. proposed a reachability algorithm on weighted graphs [19]. Reachability in distributed systems is studied in [32]. Label constrained reachability query is to judge whether there is a path between two points that only contains a subset of given labels, which is studied in [18, 31].

9 Conclusion

In this paper, we propose a new data index for solving connectivity queries in full dynamic graphs. We streamline the data structure of the state-of-the-art algorithm and reduce the time complexity of both insertion and deletion operations. We propose a new strategy to search for replacement edge in edge deletion. Furthermore, we propose a new approach that combines the advantages of spanning tree and disjoint-set tree. Our final algorithm achieves the constant query time complexity and also significantly improves the theoretical running time in both edge insertion and edge deletion. Our performance studies on real large datasets show considerable improvement in our algorithms.

Acknowledgments

Dong Wen is supported by ARC DP230101445 and ARC DE240100668. Lu Qin is supported by ARC FT200100787 and ARC DP210101347. Ronghua Li is supported by NSFC Grants U2241211 and 62072034. Ying Zhang is supported by ARC LP210301046, DP210101393 and DP230101445. Xuemin Lin is supported by NSFC U2241211, NSFC U20B2046, and GuangDong Basic and Applied Basic Research Foundation 2019B1515120048.

References

[1] Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. 1989. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record* 18, 2 (1989), 253–262.

[2] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. 2009. Incremental maintenance of 2-hop labeling of large graphs. *IEEE Transactions on Knowledge and Data Engineering* 22, 5 (2009), 682–698.

[3] Qing Chen, Oded Lachish, Sven Helmer, and Michael H. Böhlen. 2022. Dynamic Spanning Trees for Connectivity Queries on Fully-dynamic Undirected Graphs. *Proc. VLDB Endow.* 15, 11 (2022), 3263–3276.

[4] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 193–204.

[5] Cynthia Baby Daniel, S Saravanan, and Samson Mathew. 2020. Gis based road connectivity evaluation using graph theory. In *Transportation Research: Proceedings of CTRG 2017*. Springer, 213–226.

[6] Monika Rauch Henzinger and Valerie King. 1995. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. 519–527.

[7] Monika R Henzinger and Valerie King. 1999. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)* 46, 4 (1999), 502–516.

[8] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 1998. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, Jeffrey Scott Vitter (Ed.). ACM, 79–89. https://doi.org/10.1145/276698.276715

[9] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* 48, 4 (2001), 723–760. https://doi.org/10.1145/502090.502095

[10] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. 2023. Fully Dynamic Connectivity in $O(\log n(\log logn)^2)$ Amortized Expected Time. *TheoretiCS* 2 (2023).

[11] Raj Iyer, David Karger, Hariharan Rahul, and Mikkel Thorup. 2001. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *Journal of Experimental Algorithmics (JEA)* 6 (2001), 4–es.

[12] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 813–826.

[13] Qiuyi Lyu, Yuchen Li, Bingsheng He, and Bin Gong. 2021. DBL: Efficient Reachability Queries on Dynamic Graphs. In *Database Systems for Advanced Applications: 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11–14, 2021, Proceedings, Part II 26*. Springer, 761–777.

[14] Damiano Di Francesco Maesa, Andrea Marino, and Laura Ricci. 2016. Uncovering the bitcoin blockchain: an analysis of the full users graph. In *2016 IEEE international conference on data science and advanced analytics (DSAA)*. IEEE, 537–546.

[15] Zijun Mao, Hong Yao, Qi Zou, Weiting Zhang, Ying Dong, et al. 2021. Digital contact tracing based on a graph database algorithm for emergency management during the COVID-19 epidemic: Case study. *JMIR mHealth and uHealth* 9, 1 (2021), e26836.

[16] Batul J Mirza, Benjamin J Keller, and Naren Ramakrishnan. 2003. Studying recommendation algorithms by graph analysis. *Journal of intelligent information systems* 20 (2003), 131–160.

[17] Georgios A. Pavlopoulos, Maria Secrier, Charalampos N. Moschopoulos, Theodoros G. Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G. Bagos. 2011. Using graph theory to analyze biological networks. *BioData Min.* 4 (2011), 10.

[18] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering billion-scale label-constrained reachability queries within microsecond. *Proceedings of the VLDB Endowment* 13, 6 (2020), 812–825.

[19] Miao Qiao, Hong Cheng, Lu Qin, Jeffrey Xu Yu, Philip S Yu, and Lijun Chang. 2013. Computing weight constraint reachability in large networks. *The VLDB journal* 22, 3 (2013), 275–294.

[20] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. 2004. HOPI: An efficient connection index for complex XML document collections. In *Advances in Database Technology-EDBT 2004: 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004 9*. Springer, 237–255.

[21] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. 2005. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 360–371.

[22] Yossi Shiloach and Shimon Even. 1981. An on-line edge-deletion problem. *Journal of the ACM (JACM)* 28, 1 (1981), 1–4.

[23] Jingyi Song, Dong Wen, Lantian Xu, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. On Querying Historical Connectivity in Temporal Graphs. *Proc. ACM Manag. Data* 2, 3 (2024), 157.

[24] Robert Endre Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)* 22, 2 (1975), 215–225.

[25] Robert Endre Tarjan and Jan van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31, 2 (1984), 245–281.

[26] Robert Endre Tarjan and Uzi Vishkin. 1984. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *25th Annual Symposium onFoundations of Computer Science, 1984*. IEEE, 12–20.

[27] Mikkel Thorup. 2000. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 343–350.

[28] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2014. Reachability querying: An independent permutation labeling approach. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1191–1202.

[29] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2020. Efficiently answering span-reachability queries in large temporal graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1153–1164.

[30] Dong Wen, Bohua Yang, Ying Zhang, Lu Qin, Dawei Cheng, and Wenjie Zhang. 2022. Span-reachability querying in large temporal graphs. *VLDB J.* 31, 4 (2022), 629–647.

[31] Yuanyuan Zeng, Wangdong Yang, Xu Zhou, Guoqing Xiao, Yunjun Gao, and Kenli Li. 2022. Distributed Set Label-Constrained Reachability Queries over Billion-Scale Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1969–1981.

[32] Junhua Zhang, Wentao Li, Lu Qin, Ying Zhang, Dong Wen, Lizhen Cui, and Xuemin Lin. 2022. Reachability Labeling for Distributed Graphs. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 686–698.

[33] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1323–1334.

Received April 2024; revised July 2024; accepted August 2024