Dynamic Spanning Trees for Connectivity Queries on Fully-dynamic Undirected Graphs

Qing Chen  
University of Zurich  
qing@ifi.uzh.ch

Oded Lachish  
Birkbeck, University of London  
o.lachish@bbk.ac.uk

Sven Helmer  
University of Zurich  
helmer@ifi.uzh.ch

Michael H. Böhlen  
University of Zurich  
boehlen@ifi.uzh.ch

ABSTRACT

Answering connectivity queries is fundamental to fully dynamic graphs where edges and vertices are inserted and deleted frequently. Existing work proposes data structures and algorithms with worst case guarantees. We propose a new data structure, the dynamic tree (D-tree), together with algorithms to construct and maintain it. The D-tree is the first data structure that scales to fully dynamic graphs with millions of vertices and edges and, on average, answers connectivity queries much faster than data structures with worst case guarantees.

PVLDB Reference Format:  
Qing Chen, Oded Lachish, Sven Helmer, and Michael H. Böhlen. Dynamic Spanning Trees for Connectivity Queries on Fully-dynamic Undirected Graphs. PVLDB, 15(11): 3263 - 3276, 2022.  
doi:10.14778/3551793.3551868

PVLDB Artifact Availability:  
The source code, data, and/or other artifacts have been made available at https://github.com/qingchen3/D-tree.

1 INTRODUCTION

The efficient processing of large graphs is becoming ever more important (see Hegeman and Iosup [18], Sahu et al. [36], and Sakr et al. [37] for recent studies and surveys). A fundamental problem is the connectivity problem, which checks if there is a connection between two nodes in a graph. Answering connectivity queries plays a crucial role in application areas such as communication and transport networks, checking their reliability, as well as social networks, investigating the connections between users and the groups they belong to. However, it does not stop there: since dynamic connectivity is such a fundamental problem, we find applications in areas as diverse as computational geometry [12], chemistry [15], and biology [24].

Computing the connectivity between two nodes using search strategies like breadth-first search (BFS) and depth-first search (DFS) with a linear run-time is prohibitively expensive for large graphs with millions of vertices and edges. For static graphs, the connected components can be precomputed and the results stored in an auxiliary data structure, allowing the efficient processing of queries. Updating the auxiliary data structures in the fully dynamic case with frequent graph edge insertions and deletions is challenging, though. For instance, updating the well-known two-hop labeling [5, 9, 33, 52] is expensive, since BFS or DFS must be run on the graphs. Similarly, tree-based approaches [16, 22, 25, 27, 46, 49] have focused on worst-case runtime guarantees and incur high update costs for large graphs. They rely on multiple complex auxiliary data structures, have often not been implemented and evaluated empirically [3, 50], and sacrifice average case performance to get an upper bound for the worst-case complexity. In our work, we focus on fully dynamic large real-world graphs with the goal of developing a connectivity algorithm with a good average case performance for queries and updates.

First, we define what optimizing the average case complexity for connectivity queries over the spanning forest (i.e., sets of spanning trees) of a graph means: the costs are minimized if \( S_d \), the sum of distances between the root nodes and all other nodes in the spanning trees, is minimized. Since maintaining a minimal \( S_d \) in spanning trees in a fully dynamic setting is too expensive, we propose effective and practical heuristics to keep the value of \( S_d \) of the spanning trees low. Our approach has a much better average runtime than solutions with a guaranteed worst case complexity for a broad range of real-word graphs (we demonstrate this empirically).

The most time-critical part is the search for a replacement edge when deleting an edge in a spanning tree. We prove that the cost for finding a replacement edge for an edge \( e \) is proportional to the cut number of \( e \), i.e., the number of nodes in the smaller tree after removing \( e \) (deleting an edge splits a tree into two). Moreover, we prove that the average cost of finding a replacement edge is optimal for spanning trees that minimize \( S_c \), the sum of the cut numbers for all possible edges in the spanning tree. We show that \( S_d \) and \( S_c \) are directly related to each other, i.e., optimizing one also optimizes the other.

Our main technical contribution can be summarized as follows:

- We formally define the problem of evaluating connectivity queries in fully dynamic graphs with an optimal average-case complexity.
- We introduce \( S_d \) and \( S_c \). \( S_d \) is the sum of distances between roots and all other nodes; we show that the average cost of connectivity queries is optimal for spanning forests minimizing \( S_d \). \( S_c \) is the sum of cut numbers of all edges; we...
show that the average costs for finding replacement edges is optimal if spanning trees minimize $S_c$.
- We prove that $S_d = S_c$ for spanning trees in which the root is a centroid, i.e., a node that minimizes the sum of the distances to all other nodes, allowing us to optimize the average-case costs.
- We propose a novel k-ary tree, called dynamic tree (D-tree), to represent the connected components of a graph. We define D-trees and provide efficient, heuristics-based algorithms to answer connectivity queries and maintain D-trees when inserting and deleting edges.
- We embed the graph in a set of D-trees that also maintain edges not part of the spanning forest and the size of each subtree. This information helps us to keep the average runtime of operations low.
- We conduct extensive experiments to compare D-trees with existing approaches over ten real-world datasets. The experiments confirm the efficiency of our approach and its superior average-case runtime.

2 RELATED WORK

The first efficient connectivity algorithms focused on updating spanning trees in incremental [42] and decremental [39] dynamic graphs, i.e., graphs only allowing insertions or deletions, respectively. The earliest algorithms for updating minimum spanning trees in fully dynamic undirected (weighted) graphs were developed by Spira and Pan [40], Chin and Houck [8], and Frederickson [16]. The algorithm by Spira and Pan has a complexity of $O(n)$ for insertions and $O(n^{3})$ for deletions, with $n$ being the number of vertices. Chin and Houck improve the complexity for deletions to $O(n^{2})$. Frederickson brings the complexity of insertions and deletions down to $O(\sqrt{m})$, with $m$ being the number of edges. Using a technique called sparsification, Eppstein et al. improve the complexity to $O(\sqrt{n})$ per update operation [13, 14], but without providing an implementation.

Henzinger and King represent spanning trees via Euler tours [44], resulting in elegant merging and splitting of spanning trees [20–23]. Storing, searching, and maintaining Euler tours efficiently is not trivial, though. Henzinger and King proposed the Euler Tour Tree (ET-tree) [20, 22] that maps Euler tours to balanced binary trees [3, 38] and requires several auxiliary data structures [20, 22] to keep track of information for Euler tours.

The work by Henzinger and King [20, 22] sparked a whole line of research based on hierarchical forests for dynamic connectivity. We divide the algorithms into two groups: those that minimize the worst-case costs and those that optimize the amortized costs. We first look at worst-case costs for update operations. Interestingly enough, for sparse graphs, the algorithm by Frederickson [16] (and the improvement by Eppstein [14]) is still competitive. Kapron et al. [31] proposed an algorithm with complexity $O(\log^5 n)$, but it turned out that it can produce false negatives. In 2016, Keilberg-Rasmussen et al. [32] improved the complexity to $O(\sqrt{n \log \log n / \log n})$. Henzinger and King were the first to look at amortized costs and achieve polynomial logarithmic amortized complexity. Holm at al. [25] improved the bound by adding invariants to the hierarchical forests. Orthogonal data structures, such as local trees, lazy local trees, bitmaps, and a system of shortcuts [27, 46, 49], are introduced to improve the amortized complexity. The combination of these complicated data structures makes it difficult to implement (and evaluate) these algorithms. In fact, only Henzinger-King’s algorithm HK [20, 22] was fully implemented and evaluated [3, 28, 50] and is therefore our main contender.

Most existing work on labeling schemes [5, 7, 29, 47, 52] requires that input graphs are directed and/or DAGs, and consequently are generally not applicable to undirected graphs. A recent data structure for labeling, called DBL [33], works for undirected graphs. However, DBL only supports insertions on graphs, and constructing DBL is expensive since it needs to run BFS on connected components.

3 PRELIMINARIES

We consider undirected unweighted simple graphs $G(V, E)$ defined by a set of vertices $V$ and a set $E$ of edges [17, 48]. A graph is simple if there is at most one edge $(u, v) \in E$ that connects a pair of vertices $u, v \in V$. We measure the size of a graph in the number of vertices it contains, which we denote by $|V|$. Given a graph $G(V, E)$, a path $P$ is a sequence of distinct vertices $(v_1, v_2, \ldots, v_n), v_i \in V$, such that each pair of adjacent vertices in $P$, $v_i$ and $v_{i+1}$, are connected via an edge $(v_i, v_{i+1}) \in E$. The length $|P|$ of a path $P$ is defined by the number of edges in the path, i.e., for $P = (v_1, v_2, \ldots, v_n)$, $|P| = n - 1$. If there is an additional edge between $v_n$ and $v_1$, then the sequence $(v_1, v_2, \ldots, v_n)$ forms a cycle. The diameter of a graph is the length of the longest shortest path between two vertices in the graph. A connected component $C(V', E')$ is a maximal subgraph of a graph $G(V, E)$, with $V' \subseteq V, E' \subseteq E$, in which all pairs of nodes are connected via a path.

Example 3.1. Figure 1 shows a graph $G_1$ with two connected components $C_1$ and $C_2$.

![Figure 1: $G_1 = \{C_1, C_2\}$ with components $C_1$ and $C_2$](image)

A tree is an undirected graph in which any pair of vertices is connected by exactly one path. Thus, the vertices in a tree are all connected and the tree does not contain cycles. In a forest, any two vertices are connected by at most one path, which means that its connected components consist of trees. In a rooted tree, we designate one vertex as the root $r$ of the tree. By definition, $r$ has depth 0. The depth of any other vertex $v$ is determined by its (tree) distance $d_T$ to $r$, i.e., $d_T(r, v)$ is equal to the length of the path from the root to the vertex. The height of a tree is equal to the depth of the leaf node with the maximum depth. Given a rooted tree with root $r$, the ancestors, asc($v$), of a node $v \neq r$ (does not have any ancestors) are all the nodes on the path from $v$ to $r$ except $v$. The parent of $v$ is the node $u$ on this path with depth(u) + 1 = depth(v). The children of $v$ are the nodes that have $v$ as a parent. The descendants, desc($v$), of $v$
are all nodes \( u \neq v \) for which \( v \) appears in the path from \( u \) to \( r \). The subtree rooted at \( v \) consists of \( v \) and all its descendants. The size of this subtree, denoted by \( size(v) \), is measured in the number of nodes it includes. Given a connected component \( G(V', E') \), a spanning tree \( T = (V', E_T) \), with \( E_T \subseteq E' \), is a rooted tree containing all vertices of \( C \). We use a spanning forest, consisting of a spanning tree for each component, for graphs with more than one component.

Example 3.2. Figure 2 depicts spanning forest \( F_1 \) for graph \( G_1 \) from Figure 1. \( F_1 \) is made up of spanning trees \( T_1 \) and \( T_2 \) for components \( C_1 \) and \( C_2 \), respectively. The path from \( n_5 \) to \( n_1 \) is \( (n_5, n_3, n_1) \); \( anc(n_5) = \{n_1, n_3\} \); \( desc(n_3) = \{n_5\} \); \( depth(n_3) = 1 \) and \( depth(n_5) = 2 \). The subtree rooted at \( n_3 \) consists of \( n_3 \) and its descendant \( n_5 \), and the size of this subtree is 2.

\[ T_n = (DFS), \text{ from one of the two vertices and test if the search finds the} \]
\[ \text{vertex deviation} \]
\[ m \]
\[ \text{connected components of a graph, using BFS or DFS (see, e.g., [26])}, \]
\[ \text{complexity}\]
\[ \text{algorithm, such as breadth-first search (BFS) or depth-first search (DFS), from one of the two vertices and test if the search finds the} \]
\[ \text{other node, which is prohibitively expensive for large graphs (it has} \]
\[ \text{complexity} O(|V| + |E|)). \]

For static graphs, we can determine all connected components of a graph, using BFS or DFS (see, e.g., [26]), and then label the nodes with the ID of the component they belong to. Given two nodes, we then directly decide in constant time whether they are connected. Evaluating connectivity queries on dynamic graphs is a much more challenging scenario. We first formally define dynamic graphs:

**Definition 4.3 (Fully dynamic graph).** In a fully dynamic graph \( G_d(V, E) \), edges are inserted and deleted one at a time. We apply a sequence of update operations to a graph, \((t_1, o_1), (t_2, o_2), (t_3, o_3), \ldots \) where \( t_i \) is a timestamp and \( o_i \) is either an insertion \((E_{t+1} = E_t \cup \{v_i, v_j\}) \) or a deletion \((E_{t+1} = E_t \setminus \{v_i, v_j\}) \) of an edge.

Since we only deal with dynamic graphs from here on, we drop the subscript \( d \) and refer to dynamic graphs as \( G(V, E) \). Our implementation allows the insertion and deletion of isolated, i.e., unconnected vertices. However, since spanning trees consisting of a single node are trivial to handle, we restrict our description to edge insertions and deletions.

As we will see later, in the worst case the performance of deletion operations is especially problematic. We argue that these cases rarely occur in real-world graphs and that it is more important to consider the average-case complexity.

Before going into the implementation details of our approach, which is based on spanning trees, we explicitly define the problem we are solving in Definition 4.4 and then investigate important aspects of applying spanning trees to evaluate connectivity queries in fully dynamic graphs and show how we exploit these properties in the following section.

**Definition 4.4 (Problem definition).** Find a data structure that in fully dynamic graphs, on average, allows us to (a) answer connectivity queries and (b) maintain the data structure efficiently.

5 LEVERAGING SPANNING TREES

We first define the problem of evaluating connectivity queries with an optimal average-case complexity. Next, we introduce \( S_d \), which optimizes average costs for connectivity queries, and \( S_c \), which optimizes average costs for searching for replacement edges. Finally, we formally establish the relationship between \( S_c \) and \( S_d \). All proofs for the theorems and lemmas in this section are included in the technical report [6].

5.1 Evaluating Queries

We use a spanning forest to answer connectivity queries \( conn(u, v) \) by traversing the paths from \( u \) and \( v \) to the respective roots \( r_u \) and \( r_v \) of their spanning trees. If we end up at the same root, then \( u \) and \( v \) are located in the same component and are connected. If we reach different roots, they are not connected. The costs for evaluating a connectivity query \( conn(u, v) \) via spanning trees is equal to the sum of distances of \( u \) and \( v \) to their roots: \( d_T(r_u, u) + d_T(r_v, v) \).

**Definition 5.1 (Sum of distances between root and its descendants).** Given a (spanning) tree \( T = (V', E_T) \) with root \( r \), the sum of distances between \( r \) and its descendants, \( S_d \), is defined as follows:

\[ S_d(T) = \sum_{x \in V'} d_T(r, x). \]  

Before analyzing the average-case costs, we give a formal definition of these costs:
Definition 5.2 (Average-case complexity). Let $i$ be the set of all possible inputs for an algorithm $A$ and let $t(i)$, $i \in I$, be the cost of running $A$ on input $i$. The probability that input $i$ occurs is defined by $p(i)$. The average cost of running $A$ is the expected value of the running times: $E(t) = \sum_{i \in I} t(i) p(i)$. If the probabilities $p(i)$ are not available, often a uniform distribution is assumed: $E(t) = \frac{1}{|I|} \sum_{i \in I} t(i)$.

A workload-aware analysis utilizing the probability distribution of the inputs is beyond the scope of this paper. In the following, we assume a uniform distribution of the inputs. We illustrate with an example what average-case versus worst-case costs mean for connectivity queries.

Example 5.3. Consider the spanning tree $T_1$ in Figure 3(a). Then the worst case for evaluating a connectivity query occurs if we select $T_1, n_{19}$ and $T_1, n_{20}$ as parameters, leading to a cost of $3 + 3 = 6$. Assuming a uniform distribution of inputs for connectivity queries on $T_1$, we get $2 \cdot S_d(T_1) / |T_1| = \frac{2 \cdot 3}{20} = 0.3$ for the average costs. If we balance the tree by rerooting it, we get $T'_1$ as shown in Figure 3(b). For $T'_1$ the costs are 4 in the worst case and 3.5 in the average case.

(a) Structure of $T_1$, $S_d = 25$. (b) Balanced trees $T'_1$, $S_d = 35$.

Figure 3: Unbalanced versus balanced spanning trees

In Example 5.3, by balancing the spanning trees (and optimizing the worst case), we actually worsen the average costs. Looking at $T_1$ in Figure 3(a), we can see that the paths from $n_1$ to $n_{19}$ and from $n_1$ to $n_{20}$ are outliers, all the other nodes are very close to $n_1$. In essence, balancing the tree punishes the performance of all other queries not involving these outliers. For this reason, other (tree-like) data structures, such as tries [41] and multilevel extendible hashing schemes [19], do not strive for balance, but allow the outlier parts to grow deeper than the rest of the tree.

We now investigate what spanning trees have to look like to guarantee minimum average costs.

Theorem 5.4. The average costs of evaluating connectivity queries with spanning trees is optimal if the trees in the spanning forest minimize $S_d$.

Generally, a high fanout leads to shallow trees (B-trees are a classical example), which in turn decreases the distances between the root and other nodes. When it comes to spanning trees, using breadth-first-search (BFS) trees provides excellent fanout, minimizing $S_d$ for a given root.

**Lemma 5.6.** In a BFS-tree with root $r$ the sum of distances $S_d$ between $r$ and all other nodes is minimal.

So, we could compute the optimal BFS-tree for each component, i.e., if $P = \{\text{BFS-tree with root } v \in V'\}$ is the set of all BFS-trees with different roots for component $C = (V', E')$, we select the tree with $S_d = \min_{v \in V'} S_d(T)$. This optimizes the average cost of running connectivity queries via spanning trees. For fully dynamic graphs, it is too expensive to update these spanning trees while preserving them to be optimal BFS-trees. Instead, we switch to efficient heuristics, e.g., by picking a root that is a centroid.

5.2 Updating Spanning Trees

We distinguish two different types of edges in a connected component: those that belong to the current spanning tree representing the component, which we call tree edges, and those that do not, which we call non-tree edges.

**Definition 5.7 (Tree and non-tree edges).** Consider a connected component $C = (V', E')$ and a spanning tree $T = (V', E_T)$ for $C$. An edge $(u, v) \in E'$ is a tree edge for $C$ if $(u, v) \in E_T$, and a non-tree edge for $C$ if $(u, v) \in E' \setminus E_T$.

Example 5.8. Consider component $C_1 = (V_1, E_1)$ in Figure 1(a) and spanning tree $T_1$ for $C_1$ in Figure 2(a). In $E_1$, edges $(n_2, n_3)$, $(n_3, n_5)$ and $(n_4, n_5)$ are non-tree edges while all other edges are tree edges.

We first look at update operations that involve non-tree edges, which is the simpler case, and then move on to updates of tree edges. When we delete a non-tree edge $(u, v)$ in a connected component $C = (V', E')$, this does not affect the spanning tree and we do not have to make any changes to it (we know that all vertices in $C$ are still connected via the tree edges). Even better, if the spanning tree is an (optimal) BFS-tree, it will remain an (optimal) BFS-tree, since taking away an edge from $C$ does not add any shortcuts between nodes that could lead to a better tree.

Inserting a new non-tree edge $(u, v)$, i.e., both, $u$ and $v$, are in the same component $C$, means that the current spanning forest for $C$ is still valid. So, if we are only interested in maintaining spanning trees for the components of $G$, we would not have to modify anything. However, inserting a non-tree edge can invalidate that a spanning tree is a BFS-tree. Assume that $\text{depth}(u) + 1 < \text{depth}(v)$, then $v$ (and possibly some of its ancestors) can be reached faster through $u$ than taking the existing path from $v$ to the root of the tree. We can fix this case. We define $\Delta = \text{depth}(v) - \text{depth}(u)$. We disconnect $v$ and $(\Delta - 2)$ of its ancestors ($v$’s $(\Delta - 2)$-nd ancestor and $v$ have a distance of $(\Delta - 2)$) from the spanning tree, reroot this subtree to make $v$ the new root, and connect this subtree to $u$. The edge $(u, v)$ becomes a tree edge, while the edge previously connecting the $(\Delta - 2)$-nd ancestor to the tree becomes a non-tree edge. We now have a spanning tree that is a BFS-tree again. Note that the heuristic does not guarantee the optimality of the BFS-tree.

**Example 5.9.** Figure 4 shows an example of restoring a BFS-tree after inserting a non-tree edge $(n_5, n_8)$. Since $\text{depth}(n_8) + 1 < \text{depth}(n_5)$, $\Delta = \text{depth}(n_5) - \text{depth}(n_8) = 4 - 1 = 3$, and $\Delta - 2 = 1$, the $(\Delta - 2)$-nd ancestor of $n_5$ is $n_4$. We disconnect $n_4$ from the tree, turning $n_5$ into the root of the
subtree and connecting this subtree to $n_8$. The previous tree edge $(n_3, n_1)$ becomes a non-tree edge (not shown in Figure 4) and $(n_5, n_8)$ becomes a tree edge. While the tree in Figure 4(b) is a BFS-tree, it is not the BFS-tree with the optimal $S_d$ anymore. In Section 5.4 we show how to improve $S_d$.

(a) Inserting the (dashed) non-tree edge $(n_1, n_8)$, $S_d = 27$.  
(b) After restoring the BFS-tree, $S_d = 23$.

Figure 4: Restoring the BFS-tree.

Let us now turn to updates involving tree edges. If we insert a new edge $(u, v)$ into $G$ and discover that $u$ and $v$ are located in different components, $C_1$ and $C_2$ respectively, then we need to merge $C_1$ and $C_2$ into a single component $C_3$. Consequently, the spanning trees $T_1$ and $T_2$ currently representing $C_1$ and $C_2$ also need to be merged into a single spanning tree $T_3$. This involves rerooting one of the trees and connecting it to the other. Assume that we make $v$ the root of $T_2$, which, w.l.o.g., is the smaller tree, and then connect it via $(u, v)$ to $T_1$, making $(u, v)$ a tree edge in $T_3$. If we start with trees that are BFS-trees, the part covered by $T_1$ will still be one and the edge $(u, v)$ is on the shortest path to connect to vertices in $T_2$, which may not be a BFS-tree anymore after the rerooting. Essentially, this limits the damage we do to the smaller tree. Instead of rerooting $T_2$, we could run BFS on $T_2$ starting at node $v$ (to recreate a BFS-tree) and then connect $u$ to $v$. This entails costs of $O(|V_2| + |E_{T_1}|)$, compared to $O(\text{depth}(v))$ for rerooting the tree. The performance is the reason we opt for the rerooting, even though it does not guarantee an optimal BFS-tree (to recreate a BFS-tree) and then connect $u$ to $v$. If we can find a replacement edge after deleting a tree edge, the problem is finding this edge efficiently without searching through large parts of $T_1$ and $T_2$.

5.3 Searching for a Replacement Edge

A naive approach of searching for a replacement edge after a deletion is to run DFS or BFS on the resulting trees $T_1(V_1, E_{T_1})$ and $T_2(V_2, E_{T_2})$. This is costly for graphs containing large connected components ($O(|V_1| + |V_2| + |E_{T_1}| + |E_{T_2}|)$) implemented naively. There are some optimizations we can apply, though. We only need to search the smaller of the two trees $T_1$ and $T_2$: a replacement edge can be found from either direction. So, we could run the search on $T_1$ and $T_2$ in an interleaved fashion and immediately stop once we have completely traversed one of the trees (or have found a replacement edge). Alternatively, keeping track of the size of subtrees in a spanning tree, we could always run the search on the smaller tree.

In our approach, we create and maintain spanning trees in a way to increase the likelihood of an uneven split. We define the cut number of an edge $e \in E_T$ in a tree $T(V', E_T)$, which is the size of the smaller tree after splitting $T$ along $e$.

Definition 5.10 (Cut number). Given a tree $T(V', E_T)$ and an edge $e \in E_T$, we split $T$ into two subtrees, $T_1$ and $T_2$, by removing $e$ (every edge in a tree is a cut edge). We define the cut number of $e$ as the size of the smaller tree: $c(e) = \min(|T_1|, |T_2|)$. Let $S_e(T) = \sum_{e \in E_T} c(e)$ be the sum of cut numbers for $T$.

The search for a replacement edge after deleting a tree edge is proportional to the cut number of the edge we are deleting. Thus, assuming a uniform distribution for selecting a cut edge, the average costs of the search are equal to $\frac{5}{4} |E_T|$. These costs are minimized for spanning trees that minimize $S_e$, as $|E_T|$ is constant for any given spanning tree.

It is hard to analyze the cut number as defined in Definition 5.10, as we are summing over minimums. However, there is an alternative way to compute the cut number. We first formulate the following theorem (taken from [11, 51]), which we use for computing the cut number.

Theorem 5.11 (Centroid and size of subtrees). Let $m$ be (one of) the centroid(s) of a tree $T(V', E_T)$. Removing this centroid from the tree will create a forest consisting of trees $T_1, T_2, \ldots, T_k$. For every tree $T_i$, $1 \leq i \leq k$, $|T_1| = |T|/k$, i.e., each tree $T_i$ contains at most $1/k$ of the vertices of $T$.

Before computing the cut number of a tree, we move the root of the tree to (one of) the centroid(s) $m$. This allows us to get rid of the minimum in $S_e$, as we know that every subtree connected to $m$ contains at most half of the vertices. W.l.o.g. let $p_v$ be the parent of $v$, we go through all the edges $(p_v, v) \in E_T$. Due to Theorem 5.11, we know that the cut number of $(p_v, v)$ is equal to $\text{size}(v)$, the size of the subtree rooted at $v$. Therefore,

$$S_e(T) = \sum_{v \in V' \setminus m} \text{size}(v) \tag{2}$$

Lemma 5.12. For a tree $T(V', E_T)$ whose root $r$ is a centroid, the sum of cut numbers, $S_e(T)$, is equal to the sum of distances, $S_d(T)$.

Thus, the sums $S_e$ and $S_d$ are directly related to each other. Even better, utilizing Lemma 5.12 and Equation (2) (see Section 6 for details), we can maintain a low value for $S_e$ and $S_d$ using information
about the size of subtrees, which is much easier to maintain in a dynamic spanning tree than information about the depth of nodes.

With the next lemma we show that the BFS-spanning-tree \( T_m \) with the minimal sum of distances \( S_d \) for a component will always have a centroid as a root. For \( T_m \), the average costs for evaluating connectivity queries and searching for a replacement edge are minimized.

**Lemma 5.13.** Let \( P = \{ \text{BFS-tree with root } u | v \in V' \} \) be the set of BFS-trees for component \( C = (V', E') \). Let \( T_m(V_m, E_m) \in P \) with root \( r \) being the BFS-tree in \( P \) with minimal overall \( S_d \) for all trees in \( P \). Then \( r \) is a centroid of \( T_m \).

### 5.4 Fixing Spanning Trees

We have now identified what a spanning tree for a component has to look like in the ideal case to minimize the average costs for evaluating connectivity queries and searching for a replacement edge: it is the BFS-tree with the minimal sum of distances. Next, we have a closer look at how \( S_d \) is affected by updates. When we delete a non-tree edge in a component, the value of \( S_d \) for BFS-trees rooted at other nodes can never decrease, as we now have fewer options to expand the search frontier during BFS. So, we are on the safe side in this case.

While inserting a non-tree edge and rearranging subtrees as described in Section 5.2 keeps them BFS-trees, there might now be a BFS-spanning-tree rooted at another vertex with a smaller \( S_d \). For example, assume that a connected component \( C(V', E') \) only contains the (solid) edges of tree \( T(V', E_T) \) in Figure 4(a), i.e., \( E' = E_T \). Then we insert the (dashed) non-tree edge \((n_5, n_9)\) and restructure the tree to look as depicted in Figure 4(b). Clearly, this is a BFS-tree. However, if we construct a spanning tree by running a BFS starting from node \( n_8 \), we would get the tree \( T'(V', E_T) \) shown in Figure 5, with \( S_d(T') = 18 < 25 = S_d(T) \). Running a BFS on (all) vertices of a connected component after an insertion to find a BFS-tree with a better value for \( S_d \) is too expensive. Nevertheless, we can at least restore the centroid property, i.e., if we notice that the root \( r \) of the current spanning tree is not a centroid, we reroot it. As we have seen in Theorem 5.11, if we ever find a child \( c_j \) of the root with size greater than half of the vertices in the tree, we make \( r \) a child of \( c_j \) and get a tree with a smaller sum of distances \( S_d \). While this does not guarantee the best overall spanning tree for a component, it guarantees a tree that minimizes \( S_d \) for all trees with root \( c_j \) (see also Definition 3.3).

![Figure 5: Restoring centroid property, \( S_d = 18 \).](image)

Ending up with a subtree that contains more than half of the vertices can also happen during the insertion of a tree edge when we attach the smaller to the larger tree. Even splitting a spanning tree (in case we do not find a replacement edge) can lead to this situation. For example, if we delete edge \((n_2, n_3)\) in the tree shown in Figure 4(a) (before inserting \((n_5, n_8)\)), we end up with two BFS-spanning-trees, rooted at \( n_1 \) and \( n_8 \), respectively, with a suboptimal \( S_d \). Since the spanning trees we create tend to be flat with a high fan-out, going through all the children of the root can take considerable time. Instead, we piggyback the centroid restoration onto other operators.

Before we insert a tree or non-tree edge \((u, v)\), we have to go to the root of the tree(s) containing \( u \) and \( v \), to find out whether \((u, v)\) is a tree or non-tree edge. Thus, once we have reached the root, we check whether the child we came through on our way to the root has a size greater than one half of the size of the root after the insertion. If this is the case, we make this child the new root. Unfortunately, this does not work in the case of a deletion that splits a connected component, as we do not necessarily pass through the child at the root of the subtree containing more than half of the nodes. Therefore, we also check the size of the child we navigate through when we reach the root during the evaluation of a connectivity query. This defers the restoration of the centroid. However, as long as we do not have any connectivity query passing through this child, this has no influence on the query costs.

### 6 IMPLEMENTING SPANNING TREES

The implementation must be able to distinguish and handle tree and non-tree edges (as defined in Definition 5.7) in spanning trees. We start out by defining the neighborhood of a vertex.

**Definition 6.1 (neighborhoods).** Given a connected component \( C = (V', E') \), let \( \Gamma_C(v) \) (with \( v \in V' \)) denote the neighborhood of node \( v \), i.e., \( \Gamma_C(v) = \{ u \in V' | (u, v) \in E' \} \). Thus, \( \Gamma_C(v) \) contains all nodes in \( V' \) to which \( v \) is directly connected. Given a spanning tree \( T = (V, E_T) \) for component \( C \), the tree-edge neighborhood \( \Gamma_{C,T}(v) \) \( = \{ u \in V' | (u, v) \in E_T \} \) of node \( v \) is the set of nodes in \( \Gamma_C(v) \) that are directly connected to \( v \) via edges in \( E_T \). The non-tree-edge neighborhood \( \Gamma_{C,T}^{\text{nte}}(v) \) \( = \{ u \in V' | (u, v) \in E' \setminus E_T \} \) of node \( v \) contains all other edges in \( \Gamma_C(v) \). Thus, \( \Gamma_C(v) = \Gamma_{C,T}(v) \cup \Gamma_{C,T}^{\text{nte}}(v) \).

**Example 6.2.** Consider component \( C_1 \) in Figure 1(a), the neighborhood of vertex \( n_5 \), \( \Gamma_{C_1}(n_5) = \{ n_2, n_3, n_4 \} \). Given the corresponding spanning tree \( T_1 \) in Figure 2(a), the tree-edge neighborhood of node \( n_5 \), \( \Gamma_{C_1,T_1}(n_5) \) is \( \{ n_3 \} \), while its non-tree-edge neighborhood \( \Gamma_{C_1,T_1}^{\text{nte}}(n_5) \) is \( \{ n_2, n_4 \} \).

### 6.1 Dynamic Trees

A dynamic tree or D-tree is a spanning tree with additional information to facilitate its maintenance.

**Definition 6.3 (Dynamic tree (D-tree)).** A dynamic tree (D-tree) for a spanning tree \( T = (V', E_T) \) is a k-ary tree (with arbitrarily large \( k \)) in which each tree node has an attribute

- **key**, which acts as a unique identifier of a node
- **parent**, which is a pointer that links a node to its parent
- **children**, which is a set of pointers that connects a node to all its children

The attribute **key** identifies each node. We store both, **parent** and **children**, as we need to navigate both ways, e.g. traversing via parents for connectivity queries and via children searching for a replacement edge. We write \( p(v) \) to denote a pointer to node \( v \).
We add two more attributes for efficiency reasons:

- attribute size denoting the number of nodes found in the subtree rooted at a node.
- attribute nte storing the non-tree edge neighborhood $\Gamma_{C,T}^{nte}$ of a node (as neighbors to pointers).

Attribute size plays a crucial role when minimizing $S_d$ and $S_c$ (cf. Section 5), while nte allows us to embed the complete graph $G(V, E)$ into a D-tree forest. Not having to compute these attribute values on the fly speeds up the maintenance considerably. Adding an additional attribute to each node to indicate which root it belongs to would speed up queries, but at the price of slowing down updates. Every time we merge, split, or reroot a spanning tree, we would have to update this attribute: when merging or splitting we would need to update all the nodes in the smaller tree and when rerooting all the nodes in the whole tree.

**Algorithm 1**: reroot($n_w$)

```plaintext
input : tree node $n_w$ of D-tree with the root $r$
output: $n_w$, new root of the rerooted D-tree
1 ch = n_w; cur = n_w.parent; n_w.parent = NULL;
2 while cur != NULL do
  3 g = cur.parent
  4 cur.parent = ch
  5 remove ch from cur.children
  6 add cur to ch.children
  7 ch = cur; cur = g;
8 while ch.parent != NULL do
  9 ch.size = ch.size - ch.parent.size
10 ch.parent.size = ch.parent.size + ch.size
11 ch = ch.parent
12 return $n_w$
```

**Example 6.5.** In Figure 7, we employ reroot($n_1$) on a D-tree and show the D-tree after the reroot operation.

Figure 7: Example of reroot operation. The nte-attributes are not shown since they remain the same.

![Diagram showing the D-tree after reroot operation](image)

```
Figure 6: D-trees $D_1$ and $D_2$ for the spanning trees $T_1$ and $T_2$ of Figure 2, respectively. We show key, size (abbreviated with $s$), and nte as attributes, while parent and children are visualized using lines.
```

**Example 6.4.** Figure 6 shows D-tree $D_1$ for the spanning tree $T_1$ in Figure 2. Tree node $n_1$ is the root (so $n_1.parent =$ NULL), has three children ($n_1.children = \{p(n_2), p(n_3), p(n_4)\}$) and no non-tree-edge neighbors ($n_1.nte = \Gamma_{C,T}^{nte}(n_1) = \{\}$). The total number of nodes in the tree rooted at $n_1$ is 6 (so, $n_1.size =$ 6). The edge $(n_2, n_3)$ is an example of a non-tree edge and is stored in the nte-attributes of nodes $n_2$ and $n_3$ ($n_2.nte = \{n_3\}$ and $n_3.nte = \{n_2\}$).

The attributes parent and children capture the tree-edge neighborhood of a node: $\Gamma_{C,T}^{nte}(v) = \{v.parent \cup v.children\}$ (we use the dot notation to access attributes) while the non-tree-edge neighborhood of a node is stored in attribute nte. Embedding the complete graph $G(V, E)$ in a D-tree forest means that every vertex $v \in V$ appears as a node $n_v$ in a D-tree (in the following, we use $v$ and $n_v$ interchangeably) and every edge $(u, v) \in E$ appears in the set: $\{(u, x) | x \in (u.parent \cup u.children \cup u.nte)\}$.

### 6.2 Auxiliary Operations

Before going into the details of the D-tree operations, we introduce auxiliary operations to modify D-trees. These are needed, for example, to prepare the merging of D-trees or to restore BFS-trees or the centroid property. The first auxiliary operation, shown in Algorithm 1, is reroot. The reroot operation makes $n_w$ the new root, which results in a new D-tree. It follows the path from the new root $n_w$ to the previous root, swaps the parent/child relationship of two neighboring nodes, and updates the size-attributes of the visited nodes.

The link operation (see technical report [6] for pseudocode) takes two D-trees that are currently not connected and connects them via a new tree edge between $n_u$ (an arbitrary node in one of the D-trees) and $n_v$ (the root of the other D-tree). This means that we may have to call a reroot operation on one of the trees before linking them.

3269
size-attributes of the nodes on the path from \( n_u \) to \( r_u \) are increased by \( n_u.size \). If we encounter a node on the path from \( n_u \) to the root that contains more than half of the nodes in the merged tree, we restore the centroid property (cf. Section 5.4).

Example 6.6. Figure 8 shows the operation \( \text{link}(n_4, n_1, n_{10}) \) that attaches \( D_2 \) (see Figure 6(b)) to \( D_1 \) (see Figure 6(a)). Values of size-attributes of nodes on the path from \( n_4 \) to \( n_1 \) are increased by \( n_{10}.size = 4 \). Since \( n_4 \) contains more than half of the nodes of the merged tree, \( n_4 \) becomes the new centroid and we perform a reroot\((n_4) \) operation.

The unlink operation (see technical report [6] for pseudocode) splits a D-tree \( D \) into two parts, by removing the tree edge between node \( n_u \), which is a non-root node in \( D \), and its parent node. The size-attributes of all (former) ancestors of \( n_u \) are decreased by \( n_{10}.size \). After unlinking, \( n_u \) becomes the root of a separate D-tree, no adjustments are necessary in this tree. For example, in Figure 9(a), the unlink\((n_4) \) operation on \( D_1 \) of Figure 6 results in two D-trees.

### 6.3 Connectivity Queries

Algorithm 2 shows the pseudocode for running a connectivity query \( \text{conn}(n_u, n_v) \). As discussed in Section 5.4, this includes restoring the centroid property (line 3 and line 6).

#### Algorithm 2: \( \text{conn}(n_u, n_v) \)

**Input:** Tree nodes \( n_u \) and \( n_v \)

**Output:** True if \( n_u \) and \( n_v \) are connected, False otherwise

1. \( d_u = \text{Null} \)
2. while \( n_u.parent \neq \text{Null} \) do
   1. \( d_u = n_u; n_u = n_u.parent \)
3. if \( d_u \neq \text{Null} \) and \( d_u.size > n_u.size/2 \) then
   1. \( n_u = \text{reroot}(d_u) \)
4. \( d_v = \text{Null} \)
5. while \( n_v.parent \neq \text{Null} \) do
   1. \( d_v = n_v; n_v = n_v.parent \)
6. if \( d_v \neq \text{Null} \) and \( d_v.size > n_v.size/2 \) then
   1. \( n_v = \text{reroot}(d_v) \)
7. return \( n_u.key = n_v.key \)

### 6.4 Operations on Non-tree Edges

First, we determine if we are deleting a tree edge or a non-tree edge. Consider an edge \((u, v) \in E'\) in a connected component \( C = (V', E')\). If \( u \) and \( v \) are in a parent/child relationship in the D-tree representing \( C \), \((u, v) \) is a tree edge (which we cover in Section 6.5.2), otherwise it is a non-tree edge (and, thus, \( u \in v.n.e \) and \( v \in u.n.e \)).

#### 6.4.1 Deleting Non-tree Edges

Deleting a non-tree edge is the simplest update operation, as it does not affect the structure of the spanning tree, we merely need to update the n.e-attributes of the corresponding nodes. The pseudocode for the deletion of a non-tree edge is available in the technical report [6].

#### 6.4.2 Inserting Non-tree Edges

When inserting a new edge \((u, v) \) \((u, v \in V)\) into a graph \( G(V, E)\), we first run a connectivity query \( \text{conn}(u, v) \). If it returns ‘True’, then \( u \) and \( v \) are in the same component \( C \) and we are inserting a non-tree edge. Algorithm 3 shows the pseudocode of inserting a new non-tree edge (for details, see Section 5.2). The algorithm first determines the depths of \( n_u \) and \( n_v \) and the root of \( D \). If the difference of the depths is less than two, we just add \((n_u, n_v)\) as a non-tree edge to \( D \). Otherwise, (w.l.o.g. assume that \( \text{depth}(n_u) < \text{depth}(n_v) \)), we select the \((\Delta = 2)-\)nd ancestor of \( n_v \), and unlink this ancestor from \( D \) (line 14); we make \( h = n_4 \) the root of the resulting subtree and link this subtree to \( D \) (line 15).

#### Algorithm 3: \( \text{insert}(n_u, n_v, r) \)

**Input:** Tree nodes \( n_u \) and \( n_v \) (in the same D-tree \( D \)), \( r \) is root of \( D \)

**Output:** Updated D-tree after insertion of non-tree edge \((n_u, n_v)\)

1. determine \( \text{depth}(n_u), \text{depth}(n_v) \), and root \( r \) of \( D \)
2. if \( \text{depth}(n_u) \leq \text{depth}(n_v) \) then
   1. \( l = n_v; h = n_u \)
3. else \( l = n_u; h = n_v \)
4. \( \Delta = \text{depth}(h) - \text{depth}(l) \)
5. if \( \Delta < 2 \) then
   1. \( \text{add} \ n_v \ to \ n_u.n.e \)
   2. \( \text{add} \ n_u \ to \ n_v.n.e \)
   3. \( \text{reroot} \ (n_v) \)
   5. return \( r \)
8. else
   1. \( i = h \)
9. for \( x = 1 \ to \ \Delta - 2 \) do
   1. \( i = i.parent \)
11. \( \text{add} \ i \ to \ i.n.e \)
13. \( \text{link}(l, r, \text{reroot}(h)) \)

### 6.5 Operations on Tree Edges

#### 6.5.1 Inserting Tree Edges

We first discuss insertions of tree edges, which connect two previously unconnected D-trees. This means, that the connectivity query \( \text{conn}(n_u, n_v) \) came back with the result ‘False’. We also know the roots of the trees containing \( n_u \) and \( n_v \) now: they are \( r_u \) and \( r_v \), respectively. Algorithm 4 shows the pseudocode for inserting the tree edge \((n_u, n_v)\) (details in Section 5.2). Basically, we take the smaller tree (w.l.o.g. assume that this is the tree containing \( n_u \)), reroot it to \( n_u \), and connect it to \( n_v \). If necessary, the link operation also restores the centroid property.

#### Algorithm 4: \( \text{insert}(n_u, n_v, r_u, r_v) \)

**Input:** Tree nodes \( n_u \) and \( n_v \) and the roots \( r_u \) and \( r_v \) of the D-trees containing them

**Output:** Merged D-tree after insertion of tree edge \((n_u, n_v)\)

1. if \( r_u.size < r_v.size \) then
   1. return \( \text{link}(n_v, r_u, \text{reroot}(n_u)) \)
2. else return \( \text{link}(n_u, r_u, \text{reroot}(n_u)) \)

Example 6.7. Example for an insertion, \( \text{insert}(n_4, n_{10}, n_1, n_{10}) \) can be seen in Example 6.6. When inserting the tree edge \((n_4, n_{10})\), merging \( D_1 \) and \( D_2 \), we find that \( D_2 \) containing \( n_{10} \) has a smaller number of nodes. We conduct directly \( \text{link}(n_4, n_1, n_{10}) \) operation since \( n_{10} \) is already the root of the smaller tree, resulting the D-tree with \( n_4 \) as the centroid.
6.5.2 Deleting Tree Edges. We first unlink the tree along the parent/child edge \((n_u, n_v)\) and determine the root of the tree of the parent node (the child node is the root of the unlinked subtree). Next, we conduct a BFS on the tree edges in the smaller tree (the one rooted at \(r_1\)) to search for a replacement edge among the non-tree edges (line 4). If we do not find a replacement edge (line 5), we return the two unlinked D-trees. We fix the centroid property of the smaller tree if it is violated (line 6). If there are multiple replacement edges, we pick one as described in Section 5.2. In a replacement edge \((n_{r_2}, n_{r_1})\), \(n_{r_2}\) is located in the smaller tree created by unlinking the input tree, while \(n_{r_1}\) is located in the larger tree (the one rooted at \(r_1\)).

Algorithm 5: delete\(_{te}(n_u, n_v)\)

| input          | Nodes of \(n_u\) and \(n_v\) of deleted tree edge |
|----------------|--------------------------------------------------|
| output         | Either reconnected D-tree if replacement edge is found or two separate D-trees otherwise |
| 1 if \(n_u = n_v\).parent then \(ch = n_u\) else \(ch = n_u\) |
| 2 \((ch, r) = \text{unlink}(ch)\) |
| 3 if \(ch.size < r.size\) then \(r_s = ch; r_t = r\) else \(r_s = r; r_t = ch\) |
| 4 \(R = \{(n_{r_2}, n_{r_1}) | n_{r_2} \in \text{BFS}(r_2) \land n_{r_1} \in n_{r_2}.nte \land r_1 = \text{anc}(n_{r_2})\}\) |
| 5 if \(R = \emptyset\) then |
| 6 if exists non-root \(m\) with \(m.size > \frac{r.size}{2}\) then \(r_s = \text{reroot}(m)\) |
| 7 return \((r_s, r_t)\) |
| 8 else |
| 9 choose edge \((n_{r_2}, n_{r_1})\) \(\in R\) with minimal depth\(_{te}(n_{r_1})\) |
| 10 delete\(_{nte}(n_{r_2}, n_{r_1})\) |
| 11 return \((\text{insert}_{te}(n_{r_2}, n_{r_1}, r_s, r_t))\) |

Example 6.8. Figure 9 illustrates delete\(_{te}(n_1, n_4)\) on D\(_1\). First, we remove the subtree rooted at \(n_4\) via \(\text{unlink}(n_4)\), creating two D-trees. The D-tree with \(n_4\) as root is smaller in size, i.e., \(r_2 = n_4\) and \(r_1 = n_1\). We conduct a BFS starting at \(n_4\) to find replacement edges for the deleted tree edge \((n_1, n_4)\) and get back \(R = \{(n_4, n_5), (n_6, n_3)\}\) (line 4). We select the non-tree edge \((n_6, n_3)\) as the replacement edge since the depth of \(n_3(\approx 1)\) is smaller than the depth of \(n_5(\approx 2)\). We delete the non-tree edge \((n_6, n_3)\), and run insert\(_{te}(n_6, n_3, n_4, n_1)\).

![Diagram](image-url)

(a) After unlink\(_{te}(n_4)\)  (b) After reroot\(_{te}(n_6)\)  (c) After link\(_{te}(n_3, n_1, n_6)\)

Figure 9: Illustrations of delete\(_{te}(n_1, n_4)\) on D\(_1\).

Finally, we analyze the average case time complexity of the operators. Deleting a non-tree edge \((u, v)\) is the simplest operation: we just need to remove \(u\) and \(v\) from \(v.nte\) and \(u.nte\), respectively, which takes constant time. The average cost for all auxiliary operations, connectivity queries, and insertions of tree and non-tree edges is proportional to the average distance between roots and all the other nodes, that is \(\frac{S}{|V|}\), since all these operations involve traversing a spanning tree from a node to a root. Deleting a tree edge requires the traversal of the smaller tree and, potentially, the selection of a replacement edge. On average, the cost for traversing the smaller tree is equal to the average cut number, i.e., \(\frac{S}{|E|}\). When determining whether a non-tree edge is a replacement edge or not, we check if the node on the other side of the edge belongs to the other tree, which has costs similar to a query.

7 EXPERIMENTAL EVALUATION

7.1 Setup

Hardware and environment. All algorithms were implemented in Python 3. The experiments were conducted on a single machine with 500GB RAM, running Debian 10. All experiments were run 10 times on the same machine, showing very similar results.

Inserting and deleting edges. We start with empty graphs and insert (and delete) edges one at a time. When inserting a new edge \(e\) into the graph at time \(t_e\), we assign a survival time \(t_d^{e}\) to \(e\), i.e., the edge is deleted at time \(t_e^{e} + t_d^{e}\). If \(e\) is re-inserted while still in the graph, e.g., at time \(t_e^{e}\) (with \(t_e^{e} < t_e^{f} < t_e^{e} + t_d^{f}\)), the survival of \(e\) is extended, i.e., the deletion is rescheduled to \(t_e^{f} + t_d^{e}\).

The deletion of edges models that connections in graphs such as social or collaborative networks become inactive after some time. Due to the different granularity of time frames in the different graphs, we set \(t_d^{e}\) for five years for the Semantic Scholar (SC) dataset and to fourteen days for all other datasets.

Setup of measurements. Let \(t_s\) and \(t_e\) be the starting time and ending time for all updates we run on the graph, respectively. We examine test\(_{num}\) snapshots, or testing points, of the spanning trees, which are uniformly distributed in the period from \(t_s\) to \(t_e\). We use test\(_{frequency}\) equals \(\frac{t_e - t_s}{test_{num}}\) to define how frequently we evaluate connectivity queries. For all graphs except SC, we set test\(_{num}\) = 100, which means that every \(\frac{t_e - t_s}{100}\) steps, we run and evaluate connectivity queries. In the SC dataset, the edges are inserted on a yearly basis, so we introduce a testing point every year. For the timespan \(t_s\) to \(t_s\), we accumulate the run time of all update operations and show the average run time. There are variations in the size of the snapshots depending on the datasets. For example, the size of the snapshots of the Tech and YT datasets are close to the size of the actual dataset, while the snapshots for the SC dataset reach the same order of magnitude as the actual dataset toward the end of an experimental run.

Evaluating connectivity queries. At each testing point, we run connectivity queries for all pairs of vertices in small graphs and for 50 million uniformly distributed pairs in large graphs (as the total number of pairs in large graphs becomes impractical). We consider graphs with fewer than 10K vertices small graphs.

7.2 Datasets

Every graph in our datasets is represented by a set of edges with timestamps (the insertion time). All edges are undirected and we use \(|V|\) and \(|E|\) to denote the number of vertices and edges for a graph, respectively. We use the following ten real-world graphs for our experimental studies.
### Table 1: Characteristics of datasets.

| Name                        | $|V|$    | $|E|$    | # updates |
|-----------------------------|--------|--------|----------|
| email-dnc (DNC) [35]        | $1.9 \times 10^4$ | $3.74 \times 10^4$ | $3.2 \times 10^4$ |
| Call (CA) [36]              | $7 \times 10^3$    | $5.1 \times 10^4$  | $2.3 \times 10^4$ |
| messages (MS) [35]          | $2 \times 10^3$    | $6 \times 10^4$    | $6.3 \times 10^4$ |
| FB-FORUM (FB) [36]          | $8.99 \times 10^2$ | $3.4 \times 10^4$  | $3.8 \times 10^4$ |
| Wiki-elec (WI)              | $7.1 \times 10^3$  | $1.07 \times 10^5$ | $2.1 \times 10^5$ |
| tech-as-topology (Tech) [35]| $3.4 \times 10^4$  | $1.71 \times 10^5$ | $2.7 \times 10^5$ |
| Enron (EN) [36]             | $8.7 \times 10^3$  | $1.1 \times 10^6$  | $1.28 \times 10^6$ |
| youtube-growth (YT) [34]   | $3.2 \times 10^6$  | $1.44 \times 10^7$ | $2.47 \times 10^7$ |
| Stackoverflow (ST) [1]      | $2.6 \times 10^6$  | $6.3 \times 10^7$  | $7 \times 10^7$   |
| Semantic Scholar (SC) [4]  | $6.5 \times 10^7$  | $8.27 \times 10^9$ | $9.36 \times 10^9$ |

### 7.3 Evaluated Methods

We evaluate the performance of connectivity queries and maintenance operations for the following methods:

- our D-tree.
- $n$D-tree, a naive version of D-tree, that neither maintains the BFS-tree nor the centroid property, which makes it easier (and faster) to update. A performance gap between $n$D-trees and D-trees shows the effectiveness of the heuristics utilized in the D-tree.

- **opt**, optimal BFS tree: after each update, we run BFS over all vertices in the connected components affected by the update to determine the BFS-tree with minimal $S_d$. This shows how much our D-tree deviates from the optimal case.

- ET-tree: maintains an Euler tour (ET) [45] of a spanning tree. To guarantee the worst-case behavior for connectivity queries, the ET is mapped to a balanced binary tree [3, 22], which means that an ET-tree is not a spanning tree anymore. As a consequence, update operations become more expensive (for details, see [22]). Many of the algorithms mentioned in Section 2 are based on ET-trees, adding various optimizations to them [22, 25, 46, 49].

- HK, the algorithm by Henzinger and King [20, 22], is also based on ET-trees, adding information – in the form of a weight attribute – about the number of non-tree edges in a subtree. This allows the algorithm to terminate the search for a replacement edge early (if weight = 0 for a subtree). The early termination and a sampling scheme employed in the search achieves the reported amortized complexity. We implement HK with one edge level, as Alberts et al. have shown that this version consistently outperforms the version with multiple levels [3]. HK is the state-of-the-art algorithm, since this is the best algorithm among those with a worst-case guarantee mentioned in Section 2 that has been fully implemented and evaluated empirically.

- online BFS and DFS.

- Insertion-only algorithms: union-find algorithm [42, 43] and DBL [33].

### 7.4 Diameters of Real-world Graphs

Before comparing the different algorithmic approaches, we take a look at an important property of graphs and its impact on the performance of our D-tree, namely the diameter of graphs. Algorithms guaranteeing worst-case performance for connectivity queries, such as HK, focus on graphs with large diameters where the benefits of their approach are most pronounced. Dealing with worst-case scenarios adds considerable overhead to those algorithms. However, among 1324 real-world graphs we investigated [2] (see Figure 10a), 1185, or 89.5%, had a diameter not larger than sixteen. For graphs with small diameters, we can easily build and maintain D-trees with a high fanout and low depth (which is bounded by the diameter of the graph), thus achieving very good average-case performance for those graphs. This gives us an edge over HK in most real-world scenarios, as D-trees have a much higher fanout than the balanced binary trees employed by HK.

![Figure 10](image-url)

(a) Distribution of diameters (89.5% ≤ 16).
(b) D-tree outperforms HK when $\text{avg}_sp \leq 16.6$.

We quantify the difference between D-trees and HK by comparing their connectivity query performance for different values of $\text{avg}_sp$, the average sum of lengths of the shortest paths over all pairs of vertices in a graph ($\text{avg}_sp$ is upper-bounded by the diameter). Let $C = (V’, E’)$ be a connected component and $\text{dist}_C(u, v)$ the length of the shortest path between $u \in V’$ and $v \in V’$.

$$\text{avg}_sp(C) = \frac{1}{|V’|^2} \sum_{u < v} \text{dist}_C(u, v).$$

As $\text{avg}_sp$ (and the diameter) is expensive to compute for a given graph, we generated synthetic graphs with a central node and $N = 400$ other nodes arranged around this node. We connect $k$ line graphs, each containing $k$ vertices, to the central node: this regular structure allows us to compute $\text{avg}_sp$ (and the diameter) more efficiently. Figure 10b shows the connectivity query performance of D-trees and HK for different values of $\text{avg}_sp$. D-trees outperform HK for graphs with $\text{avg}_sp \leq 16.6$, so we expect D-trees to outperform HK for at least 89.5% of the real-world graphs from Figure 10a, due to the diameter being an upper bound for $\text{avg}_sp$.

### 7.5 Comparison with BFS/DFS

We compared the runtime of connectivity queries for D-trees with that of BFS/DFS, which acts as a baseline. The worst-case runtime complexity of BFS/DFS is $O(|V| + |E|)[10]$ and our experiments confirm that the runtime of this approach is too high for practical purposes: on average, BFS/DFS is several orders of magnitude slower than D-trees. For example, for one of the smaller graphs, WI, running connectivity queries for all pairs of vertices, which amounts to around 25 million queries, takes BFS/DFS more than
eight days to complete. In contrast, D-trees run this set of queries in 23 seconds. We ran the queries on the complete graph, i.e., we inserted all the edges without deleting any. Clearly, BFS/DFS does not have any maintenance costs, but it only took us 200ms to build the D-trees for the WI-graph from scratch.

### 7.6 Insertion-only Algorithms

Next, we compare D-trees with DBL and union-find [42, 43], which is still considered the state-of-the-art algorithm for insertion-only graphs [49]. We measured the average query and insertion performance per operator for D-trees, DBL, and union-find on the large graphs (excluding SC, as DBL took too long to construct the 2-hop labeling). The left-hand side of Figure 11 shows the time for inserting all the edges. Clearly, DBL is the slowest algorithm (even though we ran the insertions in a batch, which adds the smallest overhead) and D-trees are slightly slower than union-find. The right-hand side of Figure 11 shows the average runtime of running 50 million random connectivity queries (after inserting all the edges in a first step). Unsurprisingly, union-find is the fastest algorithm, followed by D-trees, and DBL comes in last again. DBL is slow, because it needs to run BFS for the insertions and from time to time also for queries. Although, union-find is the fastest algorithm, it is not applicable to fully dynamic graphs. It does not support deletions, as it only maintains compressed paths from nodes to roots and does not preserve connections among non-root vertices.

![Figure 11: Average run time for insertions and queries.](image)

### 7.7 Distances between Roots and Nodes

Here we confirm that the techniques we use for maintaining spanning trees, namely preserving BFS-trees (if possible to do so efficiently), considering short-cuts when inserting non-tree edges, and re-establishing the centroid property, lead to small values for $S_d$. In Figure 12, we show the value of $S_d$ for the current spanning forest for every snapshot. The upper row depicts the results for small graphs, for which we include the expensive methods opt and ET-tree. The best possible spanning forest is created by opt, which computes the optimal BFS-tree. We observe that our D-tree is very close to opt and much better than $n$Dtree, demonstrating the effectiveness of the heuristics for maintaining the spanning forest. Our D-tree also has better values for $S_d$ than the ET-tree and HK. The difference between the ET-tree and HK is minimal since both employ a treap [38] to balance the tree. The lower row of Figure 12 shows the results for large graphs and, again, our D-tree creates trees with small $S_d$ values and is able to maintain the lead over time. We do not show results for opt and ET-trees for large graphs, as these methods are very inefficient: opt spends about 10 seconds per update on the ST-graph (in contrast to less than one millisecond for D-trees) and we do around 20 million updates in total per experiment; after a couple of updates on the ST-graph, deletions on ET-trees are three orders of magnitude slower than those on D-trees. We do not show results for HK on the SC graph because HK ran for fourteen days and was not able to finish in that time.

Figure 16 in the technical report [6] gives a detailed insight into the distribution of node depths in the various trees. On average, the nodes in our D-trees are much closer to the roots. For small graphs (upper row of Figure 16), we are very close to opt. For large graphs (lower row of Figure 16), D-trees also outperform the other methods.

### 7.8 Performance for Connectivity Queries

As we have shown in Theorem 5.4, the average query costs are directly related to $S_d$. This is confirmed by our experiments on query performance in Figure 13. The results are strongly correlated to those for $S_d$ in Figure 12. The average Pearson correlation between $S_d$ and query time over all datasets is 0.904842. The upper row of Figure 13 for small graphs demonstrates that the performance of D-trees is very close to that of opt. Additionally, D-trees consistently outperform $n$D-trees, ET-trees, and HK for all graphs. $avg_{d}$, the average distances between nodes and roots, is less than ten in D-trees while $avg_{d}$ for HK is several times larger.

### 7.9 Performance for Update Operations

Figure 14 shows the run times for update operations. First, we see that HK is much slower than the other techniques (the differences are usually an order of magnitude). While balanced binary trees offer good worst-case performance, they are much deeper than D-trees. Moreover, HK does not use spanning trees but a more complex representation, adding to the overhead of update operations. Next, we compare D-trees to $n$D-trees to show the effectiveness and costs of our heuristics. When deleting non-tree edges, the differences are minimal: the overhead for preserving BFS-trees in D-trees is very small. We observe the biggest differences for inserting (tree and non-tree) edges. Since $n$D-trees do not utilize any heuristics for minimizing $S_d$, the distances between the roots and other nodes in the spanning trees tend to grow over time. This has a negative impact on insertions (and not just queries), because we have to navigate to the roots of the spanning trees to determine whether we insert a tree or non-tree edge. When deleting tree edges, there is no clear winner between D-trees and $n$D-trees. While D-trees have a smaller cut number, they search through all potential replacement edges to pick the best one (lowering $S_d$). $n$D-trees terminate the search for a replacement edge as soon as they find the first one.

### 7.10 Discussion

D-trees outperform HK in querying and inserting tree and non-tree edges, because of the smaller $S_d$ in the D-trees. The ET-trees employed by HK are shaped differently and do not represent spanning trees directly. Basically, the occurrences of nodes in an Euler tour of a spanning tree are mapped into a balanced binary tree such that the in-order traversal of this tree is the Euler tour. This makes it independent of the diameter of a graph and results in trees of
We identify two crucial parameters for optimizing connectivity queries via spanning trees in fully dynamic graphs: \( S_d \), the sum of distances between nodes in a tree and its root, and \( S_c \), the cut number of a tree. Due to the high cost of maintaining trees that minimize \( S_d \) and \( S_c \), we develop a data structure, called D-tree with heuristics to keep the values of \( S_d \) and \( S_c \) small when updating the trees. This makes the evaluation of connectivity queries and the maintenance of spanning trees more efficient. Moreover, we show that it is possible to implement our heuristics with a low overhead, i.e., we only need to know the size of each subtree in a spanning tree. Extensive experiments with real-world datasets demonstrate that our approach has a performance close to optimal BFS-trees and outperforms algorithms that guarantee worst-case complexity. For instance, maintaining D-trees is up to fifty times faster than HK and D-trees have a much better average query performance.

For future work, we plan to extend our approach for connectivity queries on (sparse) graphs with large diameters, such as road networks, by representing a connected component with multiple spanning trees. This makes the evaluation of connectivity queries and the maintenance of spanning trees more efficient. Moreover, we show that it is possible to implement our heuristics with a low overhead, i.e., we only need to know the size of each subtree in a spanning tree. Extensive experiments with real-world datasets demonstrate that our approach has a performance close to optimal BFS-trees and outperforms algorithms that guarantee worst-case complexity. For instance, maintaining D-trees is up to fifty times faster than HK and D-trees have a much better average query performance.

For future work, we plan to extend our approach for connectivity queries on (sparse) graphs with large diameters, such as road networks, by representing a connected component with multiple spanning trees to flatten them. We also want to make our approach workload-aware, i.e., adapt it to a given ratio of queries and update operations. Since our update operations are very efficient, we can afford to add some overhead in the form of further optimizations that our approach has a performance close to optimal BFS-trees and outperforms algorithms that guarantee worst-case complexity. For instance, maintaining D-trees is up to fifty times faster than HK and D-trees have a much better average query performance.

For future work, we plan to extend our approach for connectivity queries on (sparse) graphs with large diameters, such as road networks, by representing a connected component with multiple spanning trees. This makes the evaluation of connectivity queries and the maintenance of spanning trees more efficient. Moreover, we show that it is possible to implement our heuristics with a low overhead, i.e., we only need to know the size of each subtree in a spanning tree. Extensive experiments with real-world datasets demonstrate that our approach has a performance close to optimal BFS-trees and outperforms algorithms that guarantee worst-case complexity. For instance, maintaining D-trees is up to fifty times faster than HK and D-trees have a much better average query performance.
[47] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2018. Reachability Querying: An Independent Permutation Labeling Approach. The VLDB Journal 27, 1 (Feb 2018), 1–26. https://doi.org/10.1007/s00778-017-0468-3

[48] Douglas Brent West et al. 2003. Introduction to graph theory. Vol. 2. Prentice hall Upper Saddle River.

[49] Christian Wulff-Nilsen. 2013. Faster deterministic fully-dynamic graph connectivity. In Proceedings of the twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, 1757–1769.

[50] Christos D. Zaroliagis. 2002. Implementations and Experimental Studies of Dynamic Graph Algorithms. Springer-Verlag, Berlin, Heidelberg, 2290–278.

[51] Bohdan Zelinka. 1968. Medians and Peripherians of Trees. Archivum Mathematicum 4, 2 (1968), 87–95.

[52] 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 (Snowbird, Utah, USA) (SIGMOD ’14). ACM, New York, NY, USA, 1323–1334. https://doi.org/10.1145/2588555.2612181