# Minimum Spanning Tree Maintenance 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
YANG LU, University of Technology Sydney, Australia
XUEMIN LIN, Shanghai Jiaotong University, China

Minimum Spanning Tree (MST) is a fundamental structure in graph analytics and can be applied in various applications. The problem of maintaining MSTs in dynamic graphs is significant, as many real-world graphs are frequently updated. Existing studies on MST maintenance primarily focus on theoretical analysis and lack practical efficiency. In this paper, we propose a novel algorithm to maintain MST in dynamic graphs, which achieves high practical efficiency. In addition to the tree structure, our main idea is to maintain a replacement edge for each tree edge. In this way, the tree structure can be immediately updated when a tree edge is deleted. We propose algorithms to maintain the replacement edge for each tree edge by sharing the computation cost in the updating process. Our performance studies on large datasets demonstrate considerable improvements over state-of-the-art solutions.

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

Additional Key Words and Phrases: Minimum Spanning Tree, Weighted Graph, Dynamic Graph

**ACM Reference Format:**
Lantian Xu, Dong Wen, Lu Qin, Ronghua Li, Ying Zhang, Yang Lu, and Xuemin Lin. 2025. Minimum Spanning Tree Maintenance in Dynamic Graphs. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 54 (February 2025), 24 pages. https://doi.org/10.1145/3709704

## 1 Introduction

Given a weighted undirected graph, a minimum spanning tree (MST) is a subset of the edges that connects all the vertices together without any cycles and with the minimum possible total edge weight. When the graph is not connected, we have a minimum spanning forest instead of a single tree. An example of a weighted graph and its MST is presented in Figure 1. Computing the minimum spanning tree is a fundamental problem in graph analysis, and the corresponding algorithms are well studied, such as Kruskal algorithm [23] and Prim algorithm [28]. The problem has a wide range

*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; Yang Lu, University of Technology Sydney, Sydney, Australia, yang.lu-7@student.uts.edu.au; Xuemin Lin, Shanghai Jiaotong University, Shanghai, China, xuemin.lin@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2025/2-ART54
https://doi.org/10.1145/3709704

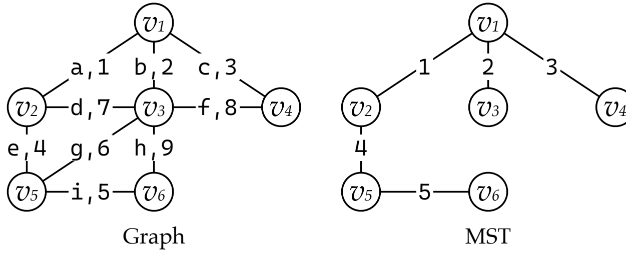



Fig. 1. A weighted graph $G$ and its MST $T$.

of applications in many fields. For instance, MST can be applied to reduce the cost of distribution network [26]. A MST for the distribution network consists of paths that connect every house without forming cycles, which minimizes the total cost of these connections. MST can be used to reflect economic relationships and analyze monetary systems [30]. In biology analysis, MST can be used to study biological membranes and wider biological structures [6]. MST is also widely studied in communication networks [21], transportation planning [2], social network analysis [3], etc.

Many real-world graphs are constantly changing and highly dynamic, where new edges are inserted and expired edges are deleted [39, 43]. The problem of maintaining MST in dynamic graphs has also been identified. The technique for MST maintenance can serve as the main subroutine for a variety of static and dynamic graph algorithms [27], such as tree packing value and edge connectivity approximation [37], dynamic k-connectivity certificate [7], dynamic minimum cut [36] and dynamic cut sparsifier [1].

**Existing Solutions.** Existing works on MST maintenance mainly aim to achieve high theoretical efficiency. General dynamic tree data structures such as ST-tree [32] and ET-tree [14] can be used to maintain MSTs. ST-tree supports inserting an edge to the graph and maintains the MST in $O(\log n)$ time complexity, where $n$ is the number of vertices in the graph. However, deleting an edge by ST-tree is more challenging. The MST is immediately split into two trees when a tree edge is deleted, and we have to identify whether a non-tree edge can replace the deleted edge and reconnect the MST. To this end, non-tree edges are scanned in a non-decreasing order of their weights, and the first one to reconnect the tree will be added to the MST. Querying whether two terminals of an edge are in the same tree based on ST-tree is $O(\log n)$, and the total time complexity to delete an edge is $O((m-n)\log n)$ as a result where $m$ is the number of edges. ET-tree can be used together with ST-tree to speed up the query efficiency, but it is still inevitable to search all non-tree edges in the worst case [4].

Holm et al. [16] adopts a top-tree data structure to preserve the MST, which achieves a time complexity of $O(\log^4 n)$ per insertion or deletion. The time complexity is further improved to $O(\log^4 n/\log\log n)$ time per operation [19]. In this paper, we refer to the algorithm of [19] as HDT, which is the state-of-the-art algorithm for MST maintenance. They mainly focus on the decremental algorithm, and the techniques can be extended to the fully dynamic setting. They divide the edges into different levels. Edges with small weights are assigned to high levels, and those with large weights are assigned to low levels. When a tree edge is deleted, they first check the local tree corresponding to the level of the deleted edge and try to find a replacement edge inside. If no replacement edge is found at the current level, they move to the lower level and repeat the process until the replacement edge is found. There are also works working on MST maintenance for batch updates, parallel and distributed algorithms [10]. We introduce them in Section 3.2.

**A Basic Framework.** To efficiently maintain MST in practice, we first show a basic method by indexing the parent of each vertex as the structure of MST. A new edge $(u, v)$ must be a tree edge if two vertices belong to different trees. If they are already in one MST, we traverse the path between

$u$ and $v$ in the tree and identify whether an existing tree edge can be replaced by $(u, v)$. A tree edge is replaced if it has the highest weight among all edges in the path and its weight is higher than that of $(u, v)$. Otherwise, the MST does not update. For edge deletion, deleting a non-tree edge would not break the correctness of the MST. However, when a tree edge is deleted, we need to search non-tree edges to identify a replacement edge for the deleted one. Searching the replacement edge is the main technical challenge in the framework. A straightforward implementation is to scan all non-tree edges in a non-decreasing order of their weights, and the replacement edge is the first one in the order that can reconnect the tree after deleting the tree edge. However, the method searches all non-tree edges in the worst case, and the overall computing cost of deleting a tree edge is very high.

**Maintaing Replacement Edges.** We improve the efficiency of deleting tree edges by maintaining the replacement edge for each tree edge. An immediate benefit is that we can directly derive the replacement edge for a deleted tree edge, while the replacement edges of certain edges need to be updated in both edge insertion and edge deletion as a result. Compared with computing the replacement edge from scratch in the basic solution, we develop novel algorithms to update replacement edges for all tree edges efficiently. We locate a small set of candidate edges whose replacement edges need updating in both edge insertion and edge deletion. For edge insertion, we observe an elegant property and guarantee the constant time complexity to update the replacement edge for each candidate edge. In this way, maintaining replacement edges does not increase the time complexity compared with the basic algorithm. For edge deletion, we update the replacement edges of candidate tree edges in a certain order. In the updating process for each candidate edge, we fully utilize the previous updated replacement edges and other update-to-date replacement edges. In this way, we share the computation cost between updating different candidate edges and significantly improve the efficiency of edge deletion.

**Contributions.** We summarize our main contributions as follows.

(a) *New techniques for maintaining replacement edges.* We design a new structure for MST maintenance by additionally indexing the replacement edge for each tree edge. We propose a novel algorithm to the replacement edges which achieves overall high efficiency in updating the MST.

(b) *Theoretical efficiency analysis.* For edge insertion, we prove that our algorithm takes $O(h)$ expected time complexity to maintain the MST, where $h$ is the average vertex depth (the distance from the vertex to the root of the tree) in the MST. For edge deletion, our algorithm takes $O(h(h + d))$ time complexity to maintain the MST, where $d$ is the average vertex degree in the graph.

(c) *Outstanding practical performance.* We conduct extensive experiments on sixteen real datasets in various settings. The results demonstrate the significantly higher practical efficiency of our method compared with the state-of-the-art solution.

**Organization.** Section 2 defines the research problem. Section 3 introduces related works and presents a basic method. Section 4 proposes our algorithms. Section 5 reports our experimental studies. Section 6 concludes the paper.

## 2 Preliminary

We study an undirected weighted graph $G(V, E)$ where $V$ and $E$ are a vertex set and an edge set, respectively. Each edge $e$ is a triplet $(u, v, w)$ where $u, v$ are two terminals and $w$ is the edge weight, i.e., $w = e.weight$. 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., $N(u) = \{v \in V | (u, v) \in E\}$. $d$ is the average degree of a vertex. Given a tree $T$ and two vertices $u, v$,

we use $path(u, v)$ to denote the path between $u$ and $v$ in $T$. We say two vertices are connected if a path containing both $u$ and $v$ exists. A tree is a connected graph without any cycle.

*Definition 2.1 (Minimum Spanning Tree).* Given a weighted graph $G$, a minimum spanning tree (MST) is a subset of edges of $G$ that connects all vertices without any cycle and with the minimum total edge weight.

A disconnected graph $G$ has a minimum spanning forest including MSTs for all connected components of $G$. For simplicity, we keep using the term MST for a graph that represents a minimum spanning forest if the graph is disconnected.

*Example 2.2.* Figure 1 shows an example of a weighted graph $G$. There are 6 vertices and 9 edges in $G$. Each edge is marked by an id for reference and a number as the weight. For example, the edge $a$ represents $(v_1, v_2)$ with a weight of 1. The MST of $G$ is shown on the right. The total edge weight of MST is 15.

**Problem Definition.** Given a weighted graph $G$, we aim to develop efficient algorithms to maintain the MST of $G$ when a new edge is inserted or an existing edge is deleted.

A dynamic weighted graph involves operations including edge insertions, edge deletions, vertex insertions, vertex deletions, and updates of edge weights. All these operations can be transferred into edge insertions and deletions. Specifically, the insertion and deletion of a vertex can be transferred into the insertion and deletion of all edges associated with that vertex, respectively. The change of the edge weight can be transferred into the deletion of an old edge followed by the insertion of a new edge. For simplicity, we assume that the weight of every edge is unique in the rest, which guarantees only one MST exists. Our algorithm can be easily extended to the scenario that multiple edges have the same weight.

## 3 Basic Methods

### 3.1 Existing Solutions

Most existing studies on MST maintenance mainly focus on improving theoretical efficiency. They can be divided into two categories. One aims to achieve better worst-case update time for every update operation [27], and the other aims to achieve better amortized update time over a series of updates [16, 19].

**Worst-Case Studies.** The initial breakthrough to address the dynamic MST problem dates back to Frederickson's 1985 algorithm, boasting a worst-case update time of $O(\sqrt{m})$ [9]. Based on their method, an $O(\sqrt{n})$ update time can be achieved by applying Eppstein et al.'s 1992 sparsification technique [7]. Christian proposed a Las Vegas data structure for fully-dynamic MST which w.h.p. handles an update in $O\left(n^{1/2-c}\right)$ worst-case time while $c < 0$ [41].

**Amortized Studies.** Henzinger and King provided an $O(n^{1/3} \log n)$ amortized update time for dynamic MST [13]. Holm et al. [16] introduced a deterministic data structure for decremental MST with an amortized update time of $O(\log^2 n)$. By combining this with a slightly adjusted version of the reduction from fully-dynamic to decremental MST as proposed by Henzinger and King [12], they achieved an overall bound of $O(\log^4 n)$ for fully-dynamic MST. Holm et al. further improved the update time to reach $O(\frac{\log^4 n}{\log \log n})$ [19].

Extensive experiments for dynamic MST algorithms were conducted by Cattaneo et al [4]. They also propose a simple but practically efficient algorithm without strong theoretical guarantee. Recently, Hanauer further reviewed the studies on dynamic MST algorithms [10]. Algorithms for batch updates are studied in [25]. There also exist parallel or distributed algorithms for MST [8, 24, 38]. [8] proposed parallel methods to compute MSTs in static directed graphs. However,

this algorithm does not support dynamic updates. Parallel algorithms for MST maintenance are only studied in some theoretical works. They can be classified into two categories. The first type targets on batch updates. Given a batch of $k$ insertions or deletions, the batch-dynamic MST algorithm proposed in [38] runs in $O(k \log^6 n)$ expected amortized work and $O(\log^4 n)$ span with high probability. However, the algorithm does not work well for a single update. It also cannot be applied to a mixed batch of updates including both insertions and deletions (e.g., in the sliding window model). The second type targets on parallel single edge updates. A theoretical approach proposed in [24] handles an edge update in MSF using $O(\sqrt{n})$ processors and $O(\log n)$ worst-case update time, with a total of $O(\sqrt{n} \log n)$ work. However, this method is not practical because it requires too many processors for large-scale graphs. For example, the dataset "edit-enwik" we used in experiment has 50,757,444 nodes, and [24] needs 7,124 processors to update one edge for the dataset.

### 3.2 Other Related Works

**Connectivity in Dynamic Graphs.** Connectivity algorithms have been developed for updating spanning trees. Henzinger and King [11, 14] introduce a method that represents spanning trees using Euler tours [34], which includes additional information to facilitate the early termination of the search for a replacement edge. Holm et al. [15, 17] proposes a novel structure that improves the update efficiency of the index to $O(\log^2 n)$. Huang et al. [20] further reduce the theoretical time complexity to $O(\log n(\log \log n)^2)$. Recently, Chen et al. [5] introduces a new data structure based on spanning tree called D-Tree to improve the connectivity query efficiency practically. [42] further proposes constant time connectivity query algorithms on dynamic graphs.

**Historical Queries in Dynamic Graphs.** The problem of efficiently answering queries for a graph snapshot of a specific time window is called historical graph queries[22, 29]. Many studies of Historical Queries have appeared in recent years. [40] proposed an index-based solution based on the concept of two-hop cover to answer historical connectivity queries. [33] further proposed a novel index for historical connectivity queries and sliding-window connectivity queries. The problem of efficiently querying historical $k$-cores in a large temporal graph is studied in [44].

### 3.3 A Non-Trivial Baseline

To efficiently maintain MST in practice, we discuss a non-trivial baseline in this section. Given a MST, we categorize edges into two types. Edges that belong to the tree are called tree edges, and others are called non-tree edges. A straightforward structure for MST is maintaining the parent of each vertex in the tree where the tree root does not have a parent. We first consider inserting a new edge $(u, v)$, and it includes two cases. **Case A** means that the new edge is a tree edge, and **Case B** means that the new edge is a non-tree edge. The insertion algorithm aims to identify these two cases.

- **Case A.1.** $(u, v)$ is a tree edge immediately if $u$ and $v$ are in different MSTs (i.e., connected components). We call this Case A.1. To test their connectivity, we search from $u$ and $v$ to their roots, respectively, and check whether their roots are the same. The time complexity is $O(h)$, where $h$ is the average vertex depth in the MST. If the roots are different, the new edge is a tree edge, and two old MSTs are merged into one MST by the new edge.
- **Case A.2.** If $u$ and $v$ have the same MST root, the old MST is still a spanning tree of the updated graph. However, the total weight of the spanning tree may not be the minimum given the new edge. The new edge $(u, v)$ and the path between $u, v$ in the old MST form a cycle. Let $e$ be the edge with the largest weight in the cycle. If $e$ is not $(u, v)$, replacing $e$ with $(u, v)$ will produce a spanning tree with a minimum total weight. We call this Case A.2, where $(u, v)$ is a tree edge in the new MST. Let $w$ be the lowest common ancestor of $u$ and $v$. The cycle contains the path from $u$ to $w$ and the path from $v$ to $w$. Therefore, it takes $O(h)$ time complexity to identify $w$ and all

edges in the cycle. To replace $e$ with $u, v$, we cut the tree by deleting $e$, rotate the tree of $v$ to make $v$ as the root, and assign $u$ as the parent $v$. The replacement process is also bounded by $O(h)$.
- **Case B.** If $(u, v)$ is the edge with the maximum weight in the cycle, inserting $(u, v)$ would not break the correctness of the MST, and the new edge is a non-tree edge. We call this Case B.

We then discuss deleting an existing edge $(u, v)$. **Case C** means that $(u, v)$ is a tree edge in the existing MST, and **Case D** means that $(u, v)$ is a non-tree edge. We do nothing for the Case D since a MST is still valid after removing any non-tree edge. For the Case C, removing $(u, v)$ immediately disconnects the tree into two subtrees, and we need to identify if an edge exists to replace $(u, v)$ and reconnect the MST.

*Definition 3.1 (Replacement Edge).* Given a graph $G$, a MST $T$ of $G$, and a tree edge $e$, an edge $e'$ is the replacement edge of $e$ if replacing $e$ with $e'$ in $T$ produces the MST of $G$ after removing $e$.

Based on Definition 3.1, the Case C is categorized into the following two sub-cases:
- **Case C.1** means no replacement edge is found because removing $(u, v)$ disconnects the graph. We leave the two MSTs as the result.
- **Case C.2** means a replacement edge is found. We apply the similar process of the Case A.2 to reconnect the MST.

The following lemma indicates the properties of the replacement edge. The lemma is straightforward based on the definition of MST, and we omit the detailed proof.

LEMMA 3.2. *A non-tree edge $e'$ is a replacement edge of $e$ if and only if (1) $e'$ connects two trees after removing $e$, and (2) $e'$ has the minimum weight among all edges satisfying the first condition.*

*Example 3.3.* Consider the graph in Figure 1. Assume the edge $b$ is deleted. The replacement edge is $g$ because it has the smallest weight among all edges reconnecting $v_3$ to the rest of the MST.

**Searching the Replacement Edge.** Based on Lemma 3.2, a straightforward approach to find the replacement edge is iterating over all non-tree edges in non-decreasing order of their weights. The iteration terminates once an edge connecting two trees is identified, and the edge is the replacement edge. The replacement edge does not exist (Case C.1) if no edge reconnects two trees. The method takes $O(m)$ to search the replacement edge. In the worst case, all non-tree edges are scanned, which is clearly not scalable for large-scale graphs. Searching the replacement edge after deleting a tree edge is the most time-consuming operation and the main challenge in MST maintenance. Our techniques mainly focus on how to reduce the large traversal time in searching the replacement edge.

## 4 Our Approach

### 4.1 The DMST Structure

We propose a novel approach for MST maintenance. Our main idea is to maintain the non-tree replacement edge for each tree edge. An immediate benefit is the $O(1)$ time complexity to get the replacement edge after deleting any tree edge. However, after the MST updates, the replacement edges for certain tree edges may be expired. Compared with the basic solution, our improved method mainly focuses on how to efficiently update the replacement edge for each tree edge. Before discussing the updating algorithms, we formally define the data structure maintained in our improved method which is called dynamic minimum spanning tree (DMST). The data structure includes several attributes for each vertex $u$ in the tree:

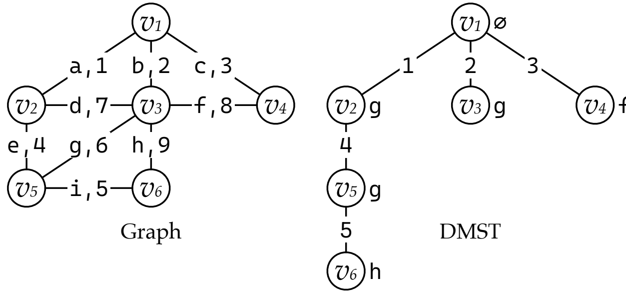



Fig. 2. An example DMST structure.

- $u.id$ // the id of the corresponding vertex;
- $u.st_size$ // the number of descendants of $u$;
- $u.parent$ // the parent of $u$;
- $u.weight$ // the weight of $(u, u.parent)$;
- $u.rep$ // the replacement edge of $(u, u.parent)$.

Given a vertex $u$ in the tree, we use $u$ to represent $u.id$ for simplicity when the context is clear. The subtree size $u.st_size$ is maintained in algorithms to reduce the average vertex depth in the tree practically. We have $u.parent = \emptyset$ if $u$ is the root of a MST. When no edge can reconnect the tree after deleting a tree edge $(u, u.parent)$, we have $u.rep = \emptyset$. We say the replacement edge of a vertex $u$ for short to represent the replacement edge of the tree edge between $u$ and $u.parent$. We maintain the replacement edge $u.rep$ as a directed edge where the descendent of $u$ in the MST is the source vertex, and the other one is the destination vertex. The following lemma demonstrates that the descendent must exists.

Lemma 4.1. *Given a vertex $u$, assume that a replacement edge $u.rep$ exists. There is one and only one vertex $v$ in $u.rep$ such that $u$ is the ancestor of $v$ in the MST ($v$ can be $u$ itself).*

Proof. The lemma is straightforward because the replacement edge reconnects the subtree of $u$ to the MST, and one terminal of the replacement edge must be in the subtree of $u$. □

We use $u.rep.src$ and $u.rep.dest$ to represent the source vertex and the destination vertex, respectively. The edge direction is used to improve the efficiency of the deletion algorithm, which will be discussed later.

*Example 4.2.* Figure 2 shows an example of DMST for the weighted graph $G$ in Figure 1. The id next to each vertex is the corresponding replacement edge $rep$. We omit $st_size$ in Figure 2. Taking $v_2$ as an example, the parent of $v_2$ is $v_1$, and the tree edge $(v_2, v_1)$ has weight 1. The edge $g$ is a replacement edge of $(v_2, v_1)$, i.e., $v_2.rep$ is $g(v_5, v_3, 6)$. $v_2$ has a total of 3 descendants, namely $\{v_2, v_5, v_6\}$.

The space complexity of our index structure is clearly bounded by the graph size. Existing algorithms [16, 19] also have linear space complexity but use relatively complex structure compared to our solution.

**Basic Operations for Tree Structure.** Based on the DMST structure, we introduce several basic operations in Algorithm 1, which will be used in our final insertion and deletion algorithms. They provide essential ways to manipulate the parent-children relationship in the tree structure, which enables us to only care if any edge should be moved between the tree edge set and the non-tree edge set in edge updates. The operations mainly focus on the tree structure and do not involve potential replacement edge updates.

**Algorithm 1:** Tree Operations

```
1  Procedure Link-Root(u, v, w)
2      u.parent ← v;
3      u.weight ← w;
4      v.st_size ← v.st_size + u.st_size;
5      u.rep ← Null;
6  Procedure Cut(u, v)
7      u.parent ← Null, u.weight ← Null, u.rep ← Null;
8      while v ≠ Null do
9          v.st_size ← v.st_size − u.st_size;
10         v ← v.parent;
11 Procedure Reroot(u)
12     if u.parent = Null then return;
13     v ← u.parent  w ← u.weight;
14     Reroot(v);
15     v.parent ← u, v.weight ← w;
16     u.parent ← Null, u.weight ← Null;
17     v.st_size ← v.st_size − u.st_size;
18     u.st_size ← v.st_size + u.st_size;
19     (x, y, w) ← u.rep;
20     u.rep ← Null;
21     v.rep ← (y, x, w);
22 Procedure Replace(u, v, w, u′, v′, w′)
23     Cut(u, v);
24     Reroot(u′), Reroot(v′);
25     if u′.st_size > v′.st_size then swap(u′, v′);
26     Link-Root(u′, v′, w′);
```

Link-Root() connects the roots of two trees and updates the subtree size accordingly. The vertex $v$ is the root of the merged tree. $w$ is the weight of edge $(u, v)$. The time complexity of Link-Root is $O(1)$. Cut() deletes an edge $(u, v)$ from the tree where $v$ is the parent of $u$. $u$ will be the root of one tree. The procedure also corrects the subtree size of every vertex from $v$ to the root given the removal of the subtree of $u$. The time complexity of Cut is $O(h)$. Reroot() does not change any tree edge. It rotates the tree to make $u$ as the tree root. The procedure runs recursively. After line 14, $v$ is the tree root. We exchange the parent-child roles of $u$ and $v$ in lines 15–16 and update the subtree size in lines 17–18. Recall that $u.rep$ in line 19 is a directed edge, where $x$ is a descendant of $u$. After the rotation, $u$ is the root, and all old descendants of $u$ are excluded from those of $v$. Therefore, $y$ is now a descendant of $v$, but $x$ is not. We reverse the direction of the replacement edge and assign it to $v.rep$ in line 21. The time complexity of Reroot is $O(h)$. Replace() replaces an existing tree edge $(u, v, w)$ with a new edge $(u', v', w')$. The tree is disconnected into two subtrees after removing $(u, v, w)$. We rotate the two trees to make $u'$ and $v'$ as roots, respectively. We merge the smaller one into the larger one.

### 4.2 Observation on Replacement Edges

We now analyze the potential change of replacement edges when updating the MST.

| | Edge Insertion Cases | | | Edge Deletion Cases | | |
|---|---|---|---|---|---|---|
| | A.1 | A.2 | B | C.1 | C.2 | D |
| Event N+ | | ✓ | ✓ | | | |
| Event N− | | | | | ✓ | ✓ |
| Event T+ | ✓ | ✓ | | | | |

Table 1. Events that trigger updating replacement edges in different cases.

Lemma 4.3. *Given the MST $T$ for a graph $G$, let $T'$ be the MST for the graph $G'$ by inserting or deleting an edge from $G$. The replacement edge of every tree edge in $T$ does not change if the non-tree edges do not change, i.e., $G \setminus T = G' \setminus T'$.*

Proof. $G \setminus T = G' \setminus T'$ happens in two cases. The first case is that a new edge is inserted, and the edge connects two existing trees. The second case is that a tree edge is deleted, and no existing edge can reconnect the trees. In both cases, the path in the tree between two terminals of any non-tree edge does not change. Therefore, the replacement edge does not update. □

Lemma 4.3 shows that updates of replacement edges are mainly caused by the update of non-tree edges. The lemma only concerns existing edges in $T$, and we also need to compute the replacement edges for any new tree edges in MST maintenance. Therefore, when edges arrive, depart, or switch between the two roles in MST maintenance, we summarize the following three events that will trigger the update of replacement edges.

- **Event** N+ represents an edge $e$ is added into non-tree edges. Certain tree edges may update their replacement edges to $e$.
- **Event** N− represents an edge $e$ is excluded from non-tree edges. The tree edges with the replacement edge of $e$ need to compute a new replacement edge.
- **Event** T+ represents an edge $e$ is added into tree edges. A replacement edge should be computed for the new tree edge $e$.

Note that we do not update replacement edges when an edge is excluded from the tree edge set, and we call this Event T−. Based on Lemma 4.3, only the Event T− would not trigger any update of replacement edges. An instance of the Event T− is the Case C.1 of edge deletion where the MST is disconnected into two MSTs. Table 1 presents the events that happened in each case of edge updates. We will discuss more details when proposing algorithms for edge insertions and deletions. We will show that the correct replacement edges can be immediately derived in edge insertions based on an elegant property. Our main technical challenge lies in edge deletions where a novel algorithm will be developed to search replacement edges.

### 4.3 Edge Insertion

In this section, we propose our final edge insertion algorithm. The pseudocode is presented in Algorithm 2. Lines 1–6 first identify whether $u$ and $v$ have the same root. If so, the loop terminates at the LCA of $u$ and $v$, where both $fu$ and $fv$ are the LCA. The edge with the maximum weight in the searching path is also recorded as a by-product (lines 4–5). When $u$ and $v$ have the same root, the searching path is the path connecting them in the tree.

**Case A.1.** Once any of $fu$ and $fv$ is Null, we already reach the root of one tree in the loop of line 2, and two vertices are not connected. We are now in Case A.1 (lines 7–10). Without any non-tree edge updates, we do not need to update the replacement edge of any existing tree edge. The new tree edge $(u, v)$ is the only one that can connect two existing MSTs. No other edge can replace $(u, v)$ if it is deleted. Therefore, we leave the replacement edge of $(u, v)$ as Null (initialized in Link-Root).

**Case B.** If $u$ and $v$ are in the same tree but $(u, v)$ has the maximum weight compared with all edges in the path between them, we fall in Case B (line 11), where the tree structure is constant. In this

**Algorithm 2:** DMST-Insert

```
   Input: a new edge e(u, v, w)
   Output: the updated MST
 1 fu ← u, fv ← v, max_e ← e;
 2 while fu ≠ Null ∧ fv ≠ Null ∧ fu ≠ fv do
 3     if fu.st_size > fv.st_size then swap(fu, fv);
 4     if fu.weight > max_e.weight then
 5         max_e ← fu.parent;
 6     fu ← fu.parent;
   /* Case A.1                                          */
 7 if fu = Null ∨ fv = Null then
 8     Reroot(u), Reroot(v);
 9     if u.st_size > v.st_size then swap(u, v);
10     Link-Root(u, v, w);
   /* Case B                                            */
11 else if max_e = e then Update-Rep(e);
   /* Case A.2                                          */
12 else
13     Replace(max_e, e);
14     Update-Rep(max_e);

15 Procedure Update-Rep(u, v, w)
16     fu ← u, fv ← v;
17     while fu ≠ fv do
18         if fu.st_size > fv.st_size then
19             swap(fu, fv), swap(u, v)
20         if fu.rep = Null ∨ w < fu.rep.weight then
21             fu.rep ← (u, v, w);
22         fu ← fu.parent;
```

case, a new non-tree edge appears as summarized in Table 1. The replacement edges of some tree edges may be updated to the new non-tree edge. Based on Lemma 3.2, $(u, v)$ is the replacement edge of a tree edge $e$ only if $e$ is in the tree path between $u$ and $v$. Therefore, to update potential replacement edges, we call Update-Rep().

Update-Rep() updates the replacement edges of all affected tree edges given a non-tree edge $(u, v, w)$. Based on the first condition of Lemma 3.2, $(u, v)$ is the replacement edge of a tree edge $e$ only if $e$ is in the tree path between $u$ and $v$. The procedure always moves from the vertex with a smaller subtree size towards the root, and this guarantees $fu$ and $fv$ meet at the lowest common ancestor of $u$ and $v$. For each edge in the path, we check if $w$ is smaller than the weight of the existing replacement edge (line 20). If so, the replacement edge is updated. The time complexity of Update-Rep is $O(h)$.

*Example 4.4.* Figure 3 shows an example of **Case B**. We insert an edge $m(v_4, v_6, 8)$ to the graph. We find $v_4$ and $v_6$ are in the same DMST in Figure 2, and there exists a $path(v_4, v_6) = \{v_6, v_5, v_2, v_1, v_4\}$ in the DMST. The weight of each edge in this path is smaller than 7. This indicates that edge $m$ is a non-tree edge, and we only need to update the $rep$ of tree edges in the path. The original $rep$ of

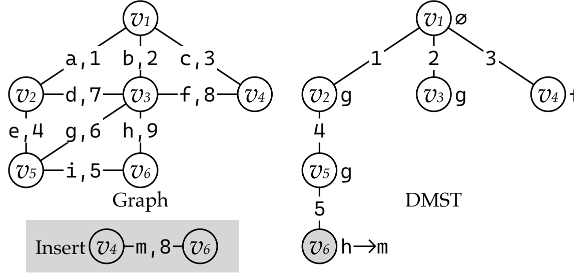



Fig. 3. Case B: Insert a non-tree edge $m(v_4, v_6, 8)$.

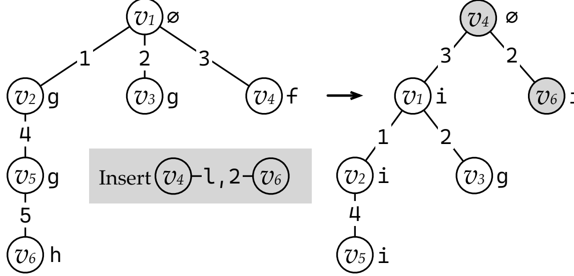



Fig. 4. Case A.2: Insert tree edge $l(v_4, v_6, 2)$.

tree edge $(v_5, v_6, 5)$ is $h$, and the *weight* of $h$ is greater than $m$. Therefore, we assign $m(v_6, v_4, 8)$ as the new *rep* of the tree edge $i(v_6, v_5, 5)$. The *rep* of other tree edges are consistent.

**Case A.2.** If $(u, v)$ is not the edge with the maximum weight (i.e., $(u, v) \neq max_e$), we fall in Case A.2, and we need to replace the existing tree edge $max_e$ with $(u, v)$. In this case, a new non-tree edge (Event N+) and a new tree edge (Event T+) appear. For the new non-tree edge, we invoke Update-Rep($max_e$) (line 15) to update the replacement edges of affected tree edges, which is similar to Case B. For the new tree edge $e$, we need to assign a replacement edge to $e$. Instead of computing the replacement edge from scratch, we can efficiently derive the replacement edge based on the following lemma.

Lemma 4.5. *Given a new edge $e$ and an existing tree edge $e'$ in MST, assume $e'$ is replaced by $e$ in the updated MST (Case A.2). The replacement edge of $e$ is $e'$.*

Proof. We prove it by contradiction. Assume the replacement edge of $e$ is $e''$, and $e'' \neq e'$. We can replace $e'$ with $e''$ in the original MST and produce a spanning tree with smaller weight. This contradicts the definition of MST. □

Lemma 4.5 indicates that the replacement edge of the new tree edge $e$ can be derived in $O(1)$ time. To this end, the procedure Replace in line 14 initializes an empty replacement edge for the new tree edge $e$, which will be assigned as $max_e$ in Update-Rep (line 21).

*Example 4.6.* Figure 4 shows an example of **Case A.2**. Given a graph $G$ and its DMST shown in Figure 2, we insert an edge $l(v_4, v_6, 2)$. We first determine the type of the edge. $v_4$ and $v_6$ are in the same DMST in Figure 2. In the path between $v_4$ and $v_6$, $i(v_6, v_5, 5)$ is the tree edge with the largest weight, and its weight is smaller than $l$. Therefore, we know $l$ will be a new tree edge. We cut the edge $i(v_6, v_5, 5)$ and use the new edge $l$ to connect $v_4$ and $v_6$. Next, we can call Update-Rep() to check if the *rep* of the tree edges in the path between $v_5$ and $v_6$ need to be updated. The *rep* of the tree edge $e(v_5, v_2, 4)$ is edge $g(v_5, v_3, 9)$. Its weight is greater than edge $i$. We can update the *rep* of the tree edge $e(v_5, v_2, 4)$ to the new non-tree edge $i$. The same update operation applies to tree edges $(v_2, v_1, 1)$, $(v_4, v_1, 3)$ and $(v_6, v_4, 2)$.

**Algorithm 3:** DMST-Delete

```
   Input: an existing edge e(u, v, w)
   Output: the updated MST
1  if st_size(u) > st_size(v) then swap(u, v);
2  if u.parent = v then
3      if u.rep = Null then
           /* Case C.1                                  */
4          Cut(u, v);
5          return;
       /* Case C.2                                      */
6      (u', v', w') ← u.rep;
7      Replace(u, v, w, u', v', w');
8      Update-Rep(u, v, w);
   /* Case D: delete non-tree edge e                    */
9  Reroot(v);
10 cand ← ∅, cand1 ← ∅, cand2 ← ∅;
11 foreach x from u to v do
12     if x.rep ≠ (u, v, w) then continue;
13     x.rep ← Null;
14     cand.push(x);
15 foreach i from 0 to |cand| − 1 do
16     x ← cand[i];
17     if st_size(x) ∗ 2 > st_size(v) then
18         foreach j from |cand| − 1 to i do
19             cand2.push(cand[j].parent);
20         Reroot(x);
21         break;
22     cand1.push(x);
23 Search(cand1, e), Search(cand2, e);
```

THEOREM 4.7. *The time complexity of Algorithm 2 is $O(h)$.*

PROOF. The time complexity of Update-Rep$(u, v, w)$ is $O(h)$. In addition, only basic operations with a time complexity of $O(h)$ are used in Algorithm 2. Therefore, the time complexity of Algorithm 2 is $O(h)$. □

### 4.4 Edge Deletion

We study the algorithm for edge deletion in this section. Our framework is presented in Algorithm 3. Lines 2–8 deal with the Case C where a tree edge is deleted. For Case C.1, we simply cut the tree in line 4. For Case C.2, our strategy first replaces the tree edge $(u, v, w)$ with its replacement edge. Then, the problem will be transferred into the Case D (i.e., deleting a non-tree edge). Specifically, we add the replacement edge $(u', v', w')$ into the tree in line 7. We update the replacement edges given the temporary new non-tree edge $(u, v, w)$. After line 8, we update replacement edges of certain tree edges given the deletion of a non-tree edge (Event N−), which happens in both Case C.2 and Case D.

**Algorithm 4:** Search

```
   Input: an order of tree nodes cand
   Output: cand with updated replacement edges
1  foreach i from 0 to |cand| − 1 do
2      D(cand[i]) ← |cand| − i;
3  foreach i from 0 to |cand| − 1 do
4      Q ← an empty priority queue;
5      u ← cand[i], urw ← +∞, Q.insert(⟨urw, u⟩);
6      while Q ≠ ∅ do
7          x ← Q.top().value, xrw ← Q.top().key, Q.pop();
8          if urw ≤ xrw then break;
9          if x ≠ u ∧ x.rep ≠ Null then
10             (s, t, w) ← x.rep;
11             if Compute-LCAD(t) > D(u) then
12                 u.rep ← x.rep;
13                 break;
14         nt_search ← True;
15         foreach ⟨y, yw⟩ ∈ N(x) do
16             if y = x.parent then continue;
17             else if x = y.parent then
18                 if y.rep = Null then continue;
19                 if y.rep.weight < urw then
20                     Q.insert(⟨y.rep.weight, y⟩);
21             else
22                 if nt_search = False then continue;
23                 if yw ≥ urw then continue;
24                 if x ≠ u ∧ xrw > yw then continue;
25                 if Compute-LCAD(y) < D(u) then
26                     u.rep ← x.rep, urw ← u.rep.weight;
27                     nt_search ← False;
28 Procedure Compute-LCAD(u)
29     x ← u;
30     while x ≠ Null do
31         if D(x) is defined then
32             D(u) ← D(x);
33             break;
34         x ← x.parent;
35     if x = Null then D(u) ← −∞;
36     x ← u;
37     while x ≠ Null do
38         if D(x) is defined then break;
39         D(x) ← D(u), x ← x.parent;
40     return D(u)
```

To delete a non-tree edge, we first compute a candidate set of tree edges whose replacement edges are expired (lines 11–14). Based on Lemma 3.2, the candidate includes all edges with the replacement edge of $e$ in the path between two terminals of the deleted non-tree edge. Note that for Case C.2, we initialize the replacement edge of the new tree edge as $e$ (line 8) so that the new tree edge will be included in the candidate set. We rotate the tree so that the number of descendants of every vertex in *cand* is less than half the size of the entire tree (lines 15–22). The main motivation of the process is to bounded the subtree size of each candidate vertex, which is related to the efficiency of searching replacement edges. A byproduct benefit is to reduce the average vertex depth of the tree, which improves certain tree operations such as Cut() and Reroot(). The new tree root partitions the candidates into two groups, and for any two candidates $u, v$ in the same group, $u$ is an ancestor or a descendant of $v$. We invoke the Search() procedure to update the replacement edges for the input candidates in line 23. The properties of the bounded number of descendants and the ancestor-descendant relationship within the group are crucial to achieve high theoretical efficiency to find the replacement edge. The bounded number of descendants will support Lemma 4.14, and the constrained vertex relationship within the same group will support Lemma 4.12.

### 4.5 Searching Replacement Edges

Given the candidate edges, the key step in edge deletion is how to compute their new replacement edges. A straightforward method is to scan all non-tree edges in non-decreasing order of their weights. We check if each edge can be the replacement edge of certain tree edges (i.e., the non-tree edge and the candidate edge form a cycle). To this end, we scan all candidate edges in the path between two terminals of each non-tree edge. The time complexity for this method is $O(mn)$ because we have $O(m)$ non-tree edges, and searching the path on the tree takes $O(n)$ time complexity.

The naive solution considers the replacement edge for each candidate tree edge independently and omits the relationship between candidate tree edges. We now propose an improved approach to compute replacement edges for a set of candidates. Given each candidate vertex $u$, the new approach searches descendants of $u$ and checks if there is an edge from a descendant to $T \setminus T(u)$.

**Cross Edge Verification.** Given a descendant $x$ and a graph neighbor $y$ of $x$, we first discuss how to identify if $y \in T \setminus T(u)$. A straightforward method is to search from $y$ to the root. If we never meet $u$, we have $y \in T \setminus T(u)$. However, we may have multiple candidate vertices and many descendants for each candidate. The straightforward method is costly. To improve the efficiency of verifying $y$, we observe that the candidate vertices are in the path between two terminals of the deleted non-tree edge. By rotating the tree (lines 15–22 of Algorithm 3), we can organize the candidates in an order such that any candidate is the ancestor of all previous candidates. This candidate structure enables sharing the computational cost of previously processed candidates. Instead of simply searching from a vertex $y$ to the root, our optimized methods compute the following concept for a vertex.

*Definition 4.8 (Lowest Candidate Ancestor Depth).* Given a set of candidate vertices $C$, the lowest candidate ancestor depth (LCAD) of a vertex $u$, denoted by $\mathcal{D}(u)$, is the depth of the first vertex $v$ in the path from $u$ to the root such that $v \in C$.

The $\mathcal{D}(u)$ of a candidate vertex $u$ itself is the depth of $u$. We set $\mathcal{D}(u) = -\infty$ if there is no candidate in the path from $u$ to the root.

LEMMA 4.9. *Let $C$ be the candidate vertices such that for any two vertices $u, v \in C$, $u$ is either an ancestor or a descendant of $v$. Given $C$, a vertex $y$ and a candidate vertex $u \in C$, we have $y \in T \setminus T(u)$ if and only if $\mathcal{D}(y) < \mathcal{D}(u)$.*

*Example 4.10.* Figure 5 shows an example of the lowest candidate ancestor depth. In this tree, $v_5$ and $v_2$ are two candidate vertices whose depths are 2 and 1, respectively. The LCAD of each vertex

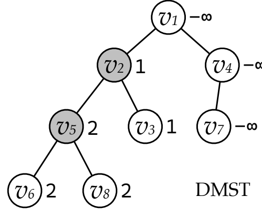



Fig. 5. An example of lowest candidate ancestor depth given the candidate vertices $\{v_2, v_5\}$.

is shown in the figure. Assume we aim to identify whether $v_3$ is in the subtree rooted by $v_5$. By comparing their LCADs, we observe that the LCAD of $v_3$ is lower than that of $v_5$, which indicates that $v_3$ is not a descendent of $v_5$.

Lemma 4.9 gives a new way to verify the neighbor of a descendant, which enables avoiding searching the same vertices. The procedure Compute-LCAD in Algorithm 4 computes the LCAD of $u$. Lines 32–37 iteratively scan the parent of $x$ to find the lowest candidate ancestor. If $x$ = Null, no candidate ancestor is found, and we set $\mathcal{D}(u)$ to $-\infty$. Based on the definition of $\mathcal{D}(u)$, the lowest candidate ancestors for all vertices from $u$ to $x$ are also $x$. Therefore, we set the LCAD of all of them as $\mathcal{D}(u)$ in lines 39–43. It is easy to see that the LCAD of any vertex is consistent given a specific candidate set. Therefore, once the LCAD of a vertex $u$ is computed, we can constantly verify $u$ for any other candidates and avoid searching from $u$ to the root.

**Pruning Search Space.** Even with a new method to verify cross edges, the descendant volume of each candidate vertex can be very large. Next, we prune by searching for unnecessary descendants by utilizing the existing replacement edges in the tree. We first give the following definition for ease of presentation.

*Definition 4.11 (Cross Edge).* Given two disjoint vertex sets $X, Y$ in a tree $T$, the cross edge, denoted by $E[X, Y]$, is a non-tree edge where two terminals are from different sets.

Given a replacement edge $e = v.rep$, we have $e \in E[T(v), T \setminus (v)]$ based on Definition 4.11. Let $C$ be the children vertices of $v$. Given that $T(v)$ contains $v$ and $T(u)$ for all $u \in C$, we divide the search space of $E[T(v), T \setminus (v)]$ into $E[\{v\}, T \setminus (v)]$ and $E[T(u), T \setminus (v)]$ for all children vertices $u$ of $v$. Searching in $E[\{v\}, T \setminus (v)]$ is straightforward by scanning neighbors of $v$. We now discuss how to search in $E[T(u), T \setminus (v)]$, and our idea is motivated by the following lemma.

Lemma 4.12. *Given a MST $T$ and two nodes $u, v$ with $u \in T(v)$, for any edge $e$ from $T(u)$ to $T \setminus T(v)$, we have $e.weight \geq u.rep.weight$.*

Proof. If exist an edge $e$ from $T(u)$ to $T \setminus T(v)$, and $e.weight < u.rep.weight$. Then $e$ is more suitable as the new $u.rep$ than $u.rep$. Therefore, such an edge $e$ does not exist. □

Lemma 4.12 implies a lower bound ($u.rep.weight$) for the edge weight in $E[T(u), T \setminus T(v)]$. Note that $u.rep$ may not be the cross edge of $[T(v), T \setminus T(v)]$, because both two terminals of $u.rep$ may be in $T(v)$. Once $u.rep$ is in $E[T(u), T \setminus T(v)]$, we have that $u.rep$ is the edge with the minimum weight in $E[T(u), T \setminus T(v)]$, and we prune searching neighbors of all vertices in $T(u)$.

*Example 4.13.* In the DMST of Figure 3, if we delete a non-tree edge $g(v_3, v_5, 6)$, we need to update the replacement edges of $e(v_2, v_5, 4)$, $a(v_1, v_2, 1)$, and $b(v_1, v_3, 2)$ in the path between $v_3$ and $v_5$ because their replacement edge is $g$. We take the edge $e(v_2, v_5, 4)$ as an example. We need to search edges from $T(v_5)$ to $T \setminus T(v_5)$ and find a new $rep$ for $e$. We divide $T(v_5)$ into $\{v_5\}$ and $T(v_6)$ because $v_5$ only has one child $v_6$. For $v_5$, there is no non-tree neighbor. For $T(v_6)$, we observe that the $rep$ of $v_6$ is $m(v_6, v_4, 8)$, which indicates $m$ is the cross edge with the minimum weight in $E[T(v_6), T \setminus T(v_6)]$. Given that $v_4$ is in $T \setminus T(v_5)$, $m$ is also the cross edge with the minimum

weight in $E[T(v_6), T \setminus T(v_5)]$, and we do not need to search any descendant of $v_6$. Edge $m$ is the final replacement edge of $e$.

**The Final Algorithm.** Based on Lemma 4.12, we propose a bottom-up algorithm to compute replacement edges of candidate vertices. The bottom-up strategy guarantees that when computing the replacement edge of a vertex $v$, the replacement edges of all descendants are available. The pseudocode of our improved searching algorithm is presented in Algorithm 4. Based on Lemma 4.9, the algorithm requires that the input candidate vertices are in a path from a vertex to the root. As mentioned in Section 4.4, lines 15–22 of Algorithm 3 guarantee the input candidate vertices of Algorithm 4 are in a path from a vertex to the root.

In Algorithm 4, we first assign the depth for each candidate. A practical optimization here is that we only compute the relative depth of candidates (lines 1–2) instead of the real distance to the root because we only compare depths between candidates in the algorithm based on Lemma 4.9.

Lines 4–27 compute the replacement edge for the candidate $u$. The variable $urw$ maintains the weight of the potential replacement edge of $u$ in iterations and will decrease to the correct value. We adopt an A* search paradigm with the weight lower bound of cross edges (Lemma 4.12). Specifically, we use a priority queue where the value of each item is a descendant $x$ of $u$, and the key is the weight of $x.rep$, which is the lower bound of the edge weight in $E[T(x), T \setminus T(u)]$. In each round of line 6, we get the item with the lowest key (weight). Once the lower bound $xrw$ is already not smaller than the current replacement edge weight $urw$ (line 8), we break the loop and $u.rep$ is already found. In line 10, we find the replacement edge of the descendant $x$ of $u$. We apply Lemma 4.12 in line 11. If the condition holds, $t$ is not in $T(u)$ so that $x.rep$ is a cross edge in $[T(u), T \setminus T(u)]$. We break the loop line line 15 since the replacement edge $u.rep$ is found.

Lines 14–27 search neighbors of $x$. If the neighbor $y$ is a child of $x$ (line 17), we consider adding $y$ to the priority queue. Line 18 means there is no edge connecting the subtree of $y$ to any other vertices outside, and searching cross edges from $y$ is pruned. $y$ is added to the priority queue only if the weight of $y.rep$ is smaller than the current replacement weight $urw$ (lines 19–20) since $y.rep.weight$ is a lower bound for those from all descendants of $y$.

Lines 22–27 search non-tree neighbors of $y$. Note that we store neighbors of each vertex in non-decreasing order of their weights. $nt_search$ in line 22 indicates if a valid cross edge (replacement edge) has been found in earlier rounds of scanning non-tree neighbors. If so, we skip searching non-tree neighbors. Line 23 indicates we already have a smaller weight of the replacement edge. In line 24, $xrw$ is the replacement edge weight of $x$. $xrw > yw$ indicates that $y$ is also in the descendant of $x$. Otherwise, $xrw$ would decrease to $yw$. Therefore, $y$ is also the descendant of $u$ and $(x, y, yw)$ cannot be the replacement edge of $u$. We check if $y$ is not in $T(u)$ in line 25. If so, we find a new potential replacement edge with a smaller weight and update the current results $u.rep$ and $urw$.

Lemma 4.14. *The expectation of the number of iterations in line 6 of Algorithm 4 is less than 2.*

Proof. According to lines 9–13 of Algorithm 4, we can terminate the search of the subtree rooted at $x$. In combination with the operation of rotating the tree in lines 15–22 of Algorithm 3, we can guarantee that the size of the subtree rooted at $x$ is less than half of the size of the whole tree (lines 9–13 of Algorithm 4). Therefore, it is more than 1/2 probability that an edge $(s, t)$ is a cross edge, i.e., the expectation of the number of iterations in line 6 of Algorithm 4 is less than 2. □

Lemma 4.15. *The complexity of the loop shown in lines 15–27 in Algorithm 4 is $O(h + d)$, where $d$ is the average degree of a vertex.*

Proof. In the loop shown in lines 15–27 of Algorithm 4, we need to traverse all neighbors of vertex $x$. Each visit can be completed in $O(1)$ if the edge visited is a tree edge. If the edge is a

| Dataset | name | $n$ | $m$ | $d$ | Type | $c$ | $wt$ | $nt$ | $h$ |
|---|---|---|---|---|---|---|---|---|---|
| advogato[1] | AD | 6,541 | 39,284 | 6.006 | Static, Trust network | 0.350 | 1.192 | 0.826 | 5.951 |
| bitcoinotc[1] | BI | 5,881 | 21,491 | 3.654 | Static, Trust network | 0.672 | 1.214 | 1.015 | 8.266 |
| retweet[2] | RE | 256,490 | 327,373 | 1.276 | Static, Social | 1.529 | 1.251 | 0.534 | 6.665 |
| orkut[1] | OR | 3,072,441 | 117,184,898 | 38.141 | Static, Social | 0.408 | 1.679 | 0.201 | 92.247 |
| amine[3] | AM | 92,830,928 | 323,836,570 | 3.488 | Static, Academic | 0.656 | 1.116 | 0.818 | 15.705 |
| road-asia-osm[2] | AS | 11,950,758 | 12,711,603 | 1.064 | Static, Road | 68.283 | 22.327 | 0.029 | 29043.425 |
| road-road-usa[2] | US | 23,947,348 | 28,854,312 | 1.205 | Static, Road | 86.452 | 103.612 | 0.077 | 29786.957 |
| spotify[3] | SP | 3,604,454 | 1,927,482,012 | 534.750 | Static, Music | 0.005 | 1.057 | 0.959 | 8.713 |
| edit-zhwikibooks[1] | BO | 14,127 | 53,881 | 3.814 | Temporal, Edit | 0.691 | 1.506 | 1.191 | 4.009 |
| edit-rowiktionary[1] | RO | 166,856 | 918,811 | 5.507 | Temporal, Edit | 0.367 | 1.111 | 0.763 | 6.918 |
| soc-flickr-growth[2] | FL | 2,302,926 | 33,140,017 | 14.390 | Temporal, Social | 0.940 | 0.232 | 0.650 | 12.258 |
| dynamic-dewiki[1] | DY | 2,166,670 | 86,337,879 | 39.848 | Temporal, Hyperlink | 0.127 | 0.001 | 0.963 | 4.896 |
| soc-bitcoin[2] | BT | 24,575,383 | 122,948,162 | 5.003 | Temporal, Transaction | 2.730 | 0.313 | 0.480 | 74.884 |
| delicious-ui[1] | UI | 34,611,304 | 301,186,579 | 8.702 | Temporal, Interaction | 0.290 | 0.065 | 0.738 | 60.230 |
| delicious-ti[1] | TI | 38,289,742 | 301,183,605 | 7.866 | Temporal, Feature | 0.285 | 0.031 | 0.681 | 20.014 |
| edit-enwiki[1] | ED | 50,757,444 | 572,591,272 | 11.281 | Temporal, Edit | 0.144 | 0.039 | 0.798 | 10.309 |

Table 2. The Description of Dataset. $d = \frac{m}{n}$ where $m$ is the number of edges and $n$ is the number of vertices. $c$ is the average size of *cand* in Algorithm 3. $wt$ is the average number of iterations in line 6 of Algorithm 4. $nt$ is the average of executions of Compute-LCAD in the loop shown in lines 17–30 in Algorithm 4. $h$ is the average vertex depth in DMST.

non-tree edge, then this visit needs to call Compute-LCAD, which can be completed in $O(h)$. For traversals of non-tree edges, it can be terminated early at line 22. Similarly to Lemma 4.14, In lines 25–27, edge$(x, y)$ lies in $E[T(x), T \setminus T(u)]$ with probability more than 1/2. Therefore, the expected number of executions of Compute-LCAD is 2. The time complexity of the entire loop shown in lines 15–27 is $O(h + d)$. □

Theorem 4.16. *The time complexity of Algorithm 3 is $O(h(h + d))$.*

Proof. In Algorithm 3, the operation of deleting tree edges utilizes only the base operations, which are $O(h)$. For the operation of deleting non-tree edges, we first traverse the path on the spanning tree of the two vertices of the deletion edge. Its complexity is $O(h)$, and the size of the resulting *cand* is also $O(h)$. Clearly, the complexity of Compute-LCAD is $O(h)$. From Lemma 4.14, the expected number of loops for line 6 of Algorithm 4 is 2. From Lemma 4.15, the complexity of the loop shown in lines 15–27 in Algorithm 4 is $O(h + d)$. Therefore, the complexity of Algorithm 4 is still $O(h + d)$. In Algorithm 3, Algorithm 4 needs to be executed $O(h)$ times, and the complexity of the whole Algorithm 3 is $O(h(h + d))$. □

Parallel processing is a common way to speed up the algorithm in practice. However, extending our algorithm to a highly parallel version is challenging given several dependencies in the updating process. For example, a key step is to find a cycle in the tree formed by a non-tree edge $(u, v)$ (e.g., inserting a new edge). Each node is identified by the parent pointer of its child node in the cycle. In Algorithm 4, searching replacement edge for deep tree nodes serves the processing of shallow tree nodes, i.e., both the updated *rep* and the LCAD contribute to updating the replacement edge for ancestor tree nodes. Therefore, a totally different computing framework is required to utilize the computing power of multi threads for MST maintenance. We will study the parallel techniques in future works.

## 5 Performance Studies

**Setup.** All algorithms are implemented by C++ and compiled with O3 level optimization. The experiments are conducted on a single machine with Intel 2.50GHz and 768GB RAM. All results are the average values by running 10 times on the same machine.

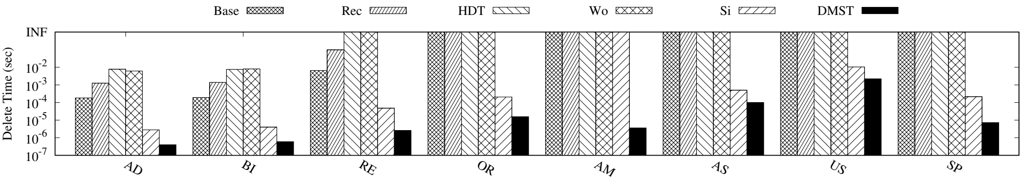



Fig. 6. Delete time of unlabeled graphs

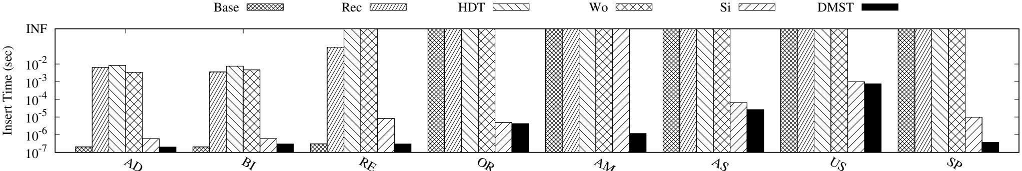



Fig. 7. Insert time of unlabeled graphs

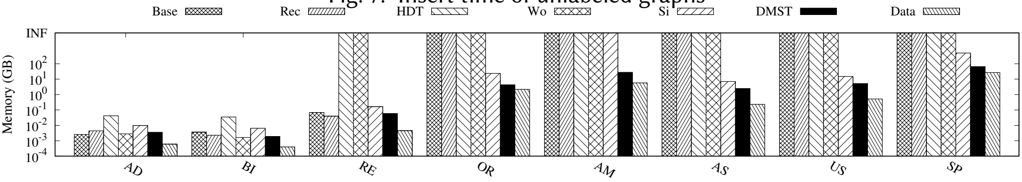



Fig. 8. Memory usage of unlabeled graphs

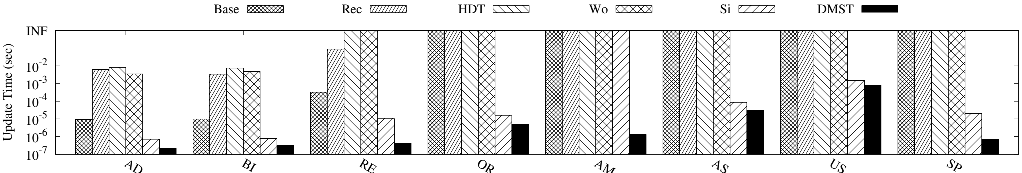



Fig. 9. Average update time (95% insertions and 5% deletions)

**Dataset.** We evaluate sixteen real datasets from different domains (Table 2). They can be found konect[1], Network Repository[2] and CORNELL[3]. Eight out of sixteen datasets are temporal graphs, and the rest are unlabeled graphs. For some graphs without weights, we randomly assign a weight to each edge.

**Competitors.** We evaluate the performance for the following methods:

- **DMST.** Our final algorithm includes all optimizations.
- **Base.** The basic algorithm discussed in Section 3.3.
- **Rec.** Computing MST from scratch for each update.
- **HDT.** The algorithm proposed by Holm et al [16, 19]. We made an implementation of the code based on [4, 18].
- **Wo.** A theoretical approach focuses on worst-case update complexity of MST [27].
- **Si.** A practical simple method proposed in [4].
- **Ba.** A method that supports batch updating of edges [25].
- **We.** A method that maintains MST, but can only support changes in edge weights [31, 35].

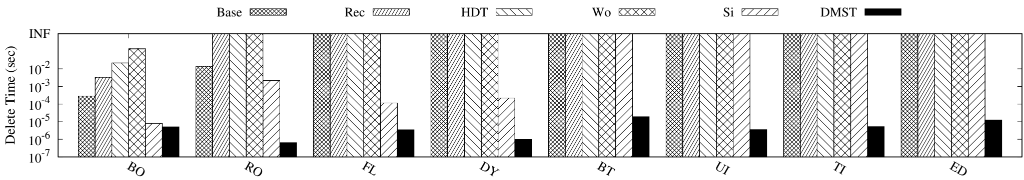



Fig. 10. Delete time of temporal graphs

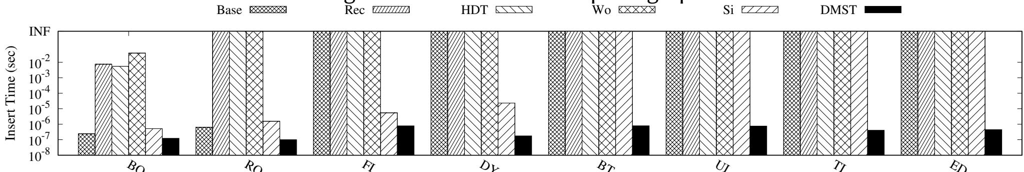



Fig. 11. Insert time of temporal graphs

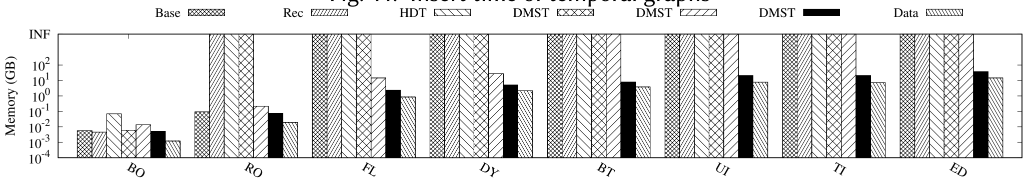
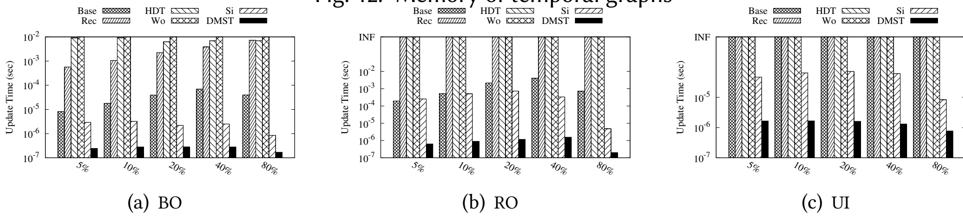
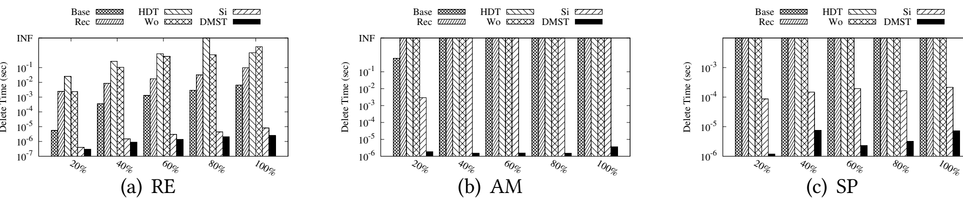



Fig. 12. Memory of temporal graphs




(a) BO (b) RO (c) UI

Fig. 13. Update time (vary window size)



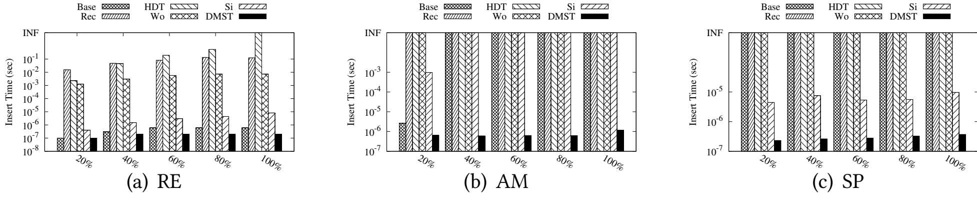

(a) RE (b) AM (c) SP

Fig. 14. Delete time (vary vertices)

## 5.1 Performance in Real Graphs

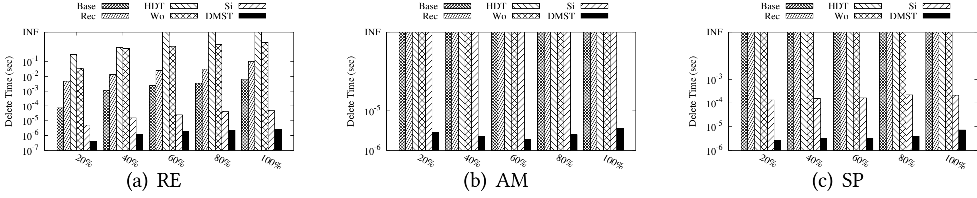
**Unlabeled graph.** For a general unlabeled graph, we first build an index with all the edges of the entire dataset. Then we randomly delete 1, 000, 000 edges and then insert those 1, 000, 000 edges back into the graph (For some small datasets, such as AD, BI and RE, the number is 10, 000). We calculate the average running time for insertions and deletions, respectively. We do not report the results of tests that take longer than 12 hours. All subsequent experiments also follow this setting.

[1] http://konect.cc/networks/
[2] https://networkrepository.com/
[3] https://www.cs.cornell.edu/~arb/data/



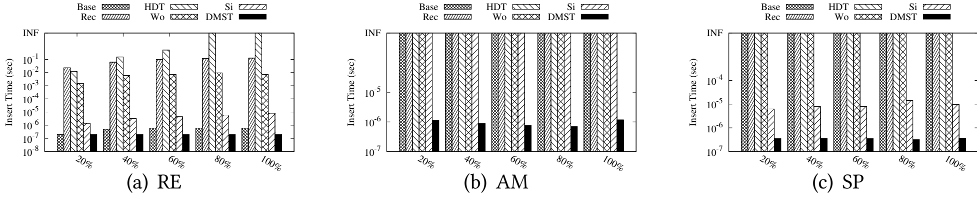

(a) RE (b) AM (c) SP

Fig. 15. Insert time (vary vertices)




(a) RE (b) AM (c) SP

Fig. 16. Delete time (vary edges)




(a) RE (b) AM (c) SP

Fig. 17. Insert time (vary edges)

Figure 6 shows the average time of the edge deletion. On all datasets, the deletion efficiency of DMST is significantly higher than that of other methods. Since DMST can efficiently find and update replacement edges, great speedups are achieved on large datasets, e.g., AM and SP. Even the time complexity of HDT looks good, but its practical efficiency is limited. In some cases, it is even worse than Rec. Wo is also not practical. Si demonstrates good deletion efficiency, especially in road networks, but still slightly worse than DMST.

Figure 7 shows the average time of edge insertion. DMST is more efficient than the other algorithms. This indicates that there is little overhead in maintaining replacement edges in the insert edge operation. Overall, insertion is more efficient than deletion for HDT and DMST. For DMST, the average insertion time is approximately an order of magnitude faster than the average deletion time. This result is exactly in line with our previous analysis of time complexity.

Figure 8 shows the memory of the six methods. HDT has the most complex structure. The memory usage of Base, Rec and DMST does not differ much from the size of the original dataset. This indicates that DMST achieves efficient update efficiency with little memory overhead.

Figure 9 shows the average update time when mixing a large number of insertions and a small number of deletions, which may be common in practice. Each dataset is updated with 10,000 edges, of which 95% are insertions and 5% are deletions. Since deletions are much slower than insertions, even a small number of deleted edges can have an impact on the overall average update speed.

**Temporal graph.** We also evaluate several temporal graphs. First, we insert all the edges one by one in the original chronological order and compute the average time of an insert operation. Finally, we continuously delete the oldest edge until the graph is empty. The average times of edge deletion and edge insertion are reported in Figure 10 and 11, respectively. Figure 12 shows the memory usage. The results are similar to unlabeled graphs. DMST is clearly superior to other algorithms.

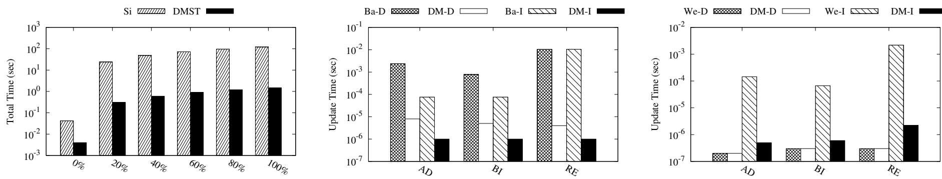



Fig. 18. Update time ($k$-clique) Fig. 19. Vertices update time Fig. 20. Edge weight update time

**Important Indicators.** Table 2 shows four important indicators in different graphs. For temporal datasets, these indicators refer to the case where the time window is set to 40%. We can see that both $wt$ and $nt$ are small and can be regarded as constants, which is consistent with Lemma 4.14 and Lemma 4.15. $c$ is also a small number. This means that for many datasets, one update does not result in a large number of replacement edges needing to be updated. Many datasets have $c$ of less than 1. The level of update efficiency is most strongly related to the average depth $h$, and this experiment proves that for most datasets, $h$ is not large.

## 5.2 Performance in Sliding Windows

We investigate six algorithms over different sizes of sliding windows in temporal graphs. For each temporal graph, we first compute the time span for the dataset. Next, 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 report three representative datasets given the space limitation.

Figure 13 illustrates the update efficiency for different window sizes on three datasets: BO, RO and UI. DMST remains far more efficient than other algorithms. As the window increases, the number of edges in a window increases. The efficiency of Base and Rec decreases as the window size gets larger. In contrast, the update efficiency of HDT and DMST remains stable.

## 5.3 Scalability Testing

In the scalability test experiment, we test the scalability of three representative large datasets (RE, AM, SP) by randomly sampling their vertices and edges from 20% to 100%. The average running time is calculated in the same way as for the unlabelled graph in Section 5.1.

**Deletion.** Figure 14 and Figure 16 show the deletion time of different scales. Among these algorithms, DMST demonstrates the optimal performance. With the increase of vertices (edges), the deletion time of the other algorithms increases, while the time of DMST is more stable.

**Insertion.** Figure 15 and Figure 17 show the average insert time of different scales. The performance of DMST is more stable and significantly better than the other algorithms. As the number of vertices (edges) grows, the insertion and deletion times show similar trends.

## 5.4 Performance in $k$-clique graph

In [4], $k$-clique graph is defined to test the update efficiency of MST maintenance algorithms when replacement edges are hard to find. Given a size threshold $c$, a $k$-clique graph contains $k$ $c$-cliques and these cliques are connected with $2k$ inter-clique edges. The inter-clique edges are assigned weights that are larger than those of the intra-clique edges, making it more challenging to find a replacement for a deleted inter-clique edge. We build a $k$-clique graph $G$ with five 200-cliques and 10 inter-clique edges. Similar as previous experiments, we delete 10,000 edges and insert the 10,000 edges back to $G$. This process is repeated six times. In each update, the percentage of inter-clique

edges is 0%, 20%, 40%, 60%, 80% and 100%, respectively. Figure 18 shows the total update time for these updates. It is clear that DMST is still works well when updating the inter-clique edges.

### 5.5 Performance of other types of updates

As we mentioned before, vertex insertions, vertex deletions, and updates of edge weights can be transferred into edge insertions and deletions. We compare our algorithm with two existing algorithms designed to update vertices and edge weights, respectively.

**Vertex updates.** We compare DMST with the method proposed in [25]. For each dataset, we randomly selected 1,000 vertices, deleted them all and reinserted them. Figure 19 shows the average update time of a vetex. "-D" means vertex deletion. "-I" means vertex insertion. DMST performs better at both deletion and insertion.

**Weight updates.** We compare DMST with the method proposed in [31] for edge weight updates. For each dataset, we randomly selected 1,000 edges. We double the weights of these edges and change them back to their original weights. Figure 20 shows the average update time of an update. "-D" indicates a decrease in weight. "-I" indicates an increase in weight. The efficiency of weight reduction in DMST is similar to that of We. DMST is much faster than We in terms of weight increase.

## 6 Conclusion

This paper proposes a new data index for maintaining minimum spanning trees in full dynamic graphs. Our main idea is to index a replacement edge for each tree edge. The structure helps identify the replacement edge in constant time when a tree edge is deleted. We propose efficient algorithms to efficiently maintain replacement edges for all tree edges in insertion and deletion. Our experimental results demonstrate the significant advantage of our algorithms on real large datasets.

## 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] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. 2016. On fully dynamic graph sparsifiers. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 335–344.

[2] NP Akpan and IA Iwok. 2017. A minimum spanning tree approach of solving a transportation problem. *International Journal of Mathematics and Statistics Invention* 5, 3 (2017), 09–18.

[3] Khawla Asmi, Dounia Lotfi, and Mohamed El Marraki. 2016. A novel approach based on the minimum spanning tree to discover communities in social networks. In *2016 International Conference on Wireless Networks and Mobile Communications (WINCOM)*. IEEE, 286–290.

[4] Giuseppe Cattaneo, Pompeo Faruolo, U Ferraro Petrillo, and Giuseppe F Italiano. 2010. Maintaining dynamic minimum spanning trees: An experimental study. *Discrete Applied Mathematics* 158, 5 (2010), 404–425.

[5] 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.

[6] C Dusser, M Rasigni, J Palmari, G Rasigni, A Llebaria, and F Marty. 1987. Minimal spanning tree analysis of biological structures. *Journal of theoretical biology* 125, 3 (1987), 317–323.

[7] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. 1997. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)* 44, 5 (1997), 669–696.

[8] Orr Fischer and Rotem Oshman. 2023. A distributed algorithm for directed minimum-weight spanning tree. *Distributed Computing* 36, 1 (2023), 57–87.

[9] Greg N Frederickson. 1983. Data structures for on-line updating of minimum spanning trees. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 252–257.
[10] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2022. Recent advances in fully dynamic graph algorithms–a quick reference guide. *ACM Journal of Experimental Algorithmics* 27 (2022), 1–45.
[11] 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.
[12] Monika Rauch Henzinger and Valerie King. 1997. Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation. *SRC Technical Note* 4 (1997).
[13] Monika R Henzinger and Valerie King. 1997. Maintaining minimum spanning trees in dynamic graphs. In *Automata, Languages and Programming: 24th International Colloquium, ICALP'97 Bologna, Italy, July 7–11, 1997 Proceedings 24*. Springer, 594–604.
[14] 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.
[15] 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
[16] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)* 48, 4 (2001), 723–760.
[17] 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
[18] Jacob Holm, Eva Rotenberg, and Alice Ryhl. 2023. Splay top trees. In *Symposium on Simplicity in Algorithms (SOSA)*. SIAM, 305–331.
[19] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. 2015. Faster fully-dynamic minimum spanning forest. In *Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*. Springer, 742–753.
[20] 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).
[21] Anjani Jain and John W Mamer. 1988. Approximations for the random minimal spanning tree with application to network provisioning. *Operations Research* 36, 4 (1988), 575–584.
[22] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 997–1008.
[23] Jon Kleinberg and Eva Tardos. 2006. *Algorithm design*. Pearson Education India.
[24] Tsvi Kopelowitz, Ely Porat, and Yair Rosenmutter. 2018. Improved worst-case deterministic parallel dynamic minimum spanning forest. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 333–341.
[25] Mao Luo, Huigang Qin, Xinyun Wu, Caiquan Xiong, Dahai Xia, and Yuanzhi Ke. 2024. Efficient Maintenance of Minimum Spanning Trees in Dynamic Weighted Undirected Graphs. *Mathematics* 12, 7 (2024), 1021.
[26] Adarsh Nagarajan and Raja Ayyanar. 2014. Application of minimum spanning tree algorithm for network reduction of distribution systems. In *2014 North American Power Symposium (NAPS)*. IEEE, 1–5.
[27] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. 2017. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 950–961.
[28] Robert Clay Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401.
[29] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment* 4, 11 (2011), 726–737.
[30] Marcel Rešovskỳ, Denis Horváth, Vladimír Gazda, and Marianna Siničáková. 2013. Minimum spanning tree application in the currency market. *Biatec* 21, 7 (2013), 21–23.
[31] Celso C Ribeiro and Rodrigo F Toso. 2007. Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes on edge weights. In *Experimental Algorithms: 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007. Proceedings 6*. Springer, 393–405.
[32] Daniel D Sleator and Robert Endre Tarjan. 1981. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. 114–122.
[33] Jingyi Song, Dong Wen, Lantian Xu, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. On Querying Historical Connectivity in Temporal Graphs. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–25.

[34] 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.

[35] Robert E Tarjan and Renato F Werneck. 2010. Dynamic trees in practice. *Journal of Experimental Algorithmics (JEA)* 14 (2010), 4–5.

[36] Mikkel Thorup. 2007. Fully-dynamic min-cut. *Combinatorica* 27, 1 (2007), 91–127.

[37] Mikkel Thorup and David R Karger. 2000. Dynamic graph algorithms with applications. In *Scandinavian Workshop on Algorithm Theory*. Springer, 1–9.

[38] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2022. Parallel batch-dynamic minimum spanning forest and the efficiency of dynamic agglomerative graph clustering. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 233–245.

[39] Xubo Wang, Dong Wen, Wenjie Zhang, Ying Zhang, and Lu Qin. 2023. Distributed Near-Maximum Independent Set Maintenance over Large-scale Dynamic Graphs. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2538–2550.

[40] 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.

[41] Christian Wulff-Nilsen. 2017. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. 1130–1143.

[42] Lantian Xu, Dong Wen, Lu Qin, Ronghua Li, Ying Zhang, and Xuemin Lin. 2024. Constant-time Connectivity Querying in Dynamic Graphs. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–23.

[43] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Xubo Wang, and Xuemin Lin. 2019. Fully Dynamic Depth-First Search in Directed Graphs. *Proc. VLDB Endow.* 13, 2 (2019), 142–154.

[44] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On Querying Historical K-Cores. *Proc. VLDB Endow.* 14, 11 (2021), 2033–2045.


Received July 2024; revised September 2024; accepted November 2024