Fully Dynamic Rooted Spanning Tree on GPU

Anonymous Author(s)

Abstract

Spanning trees are fundamental structures in graph theory, essential for various applications such as network maintenance, routing adjustments, and many more. The dynamic nature of real-world networks requires efficient updates to these structures as the underlying graph evolves. Maintaining rooted spanning trees dynamically is particularly crucial for algorithms addressing 2-connected components and minimum-weighted spanning trees. In this paper, we address the challenge of maintaining a rooted spanning forest when a batch of edges are inserted or deleted. We present four novel fully dynamic parallel algorithms to update the spanning forest without reconstructing it from scratch. To the best of our knowledge, parallel algorithms for this problem remain largely unexplored. Our experiments on a diverse collection of real-world graphs using a GPU environment demonstrate a throughput of 2 million insertions and 1.4 million deletions per second, significantly outperforming state-of-the-art parallel static algorithms.

CCS Concepts

• **Theory of computation → Massively parallel algorithms**; **Dynamic graph algorithms**.

Keywords

Dynamic graph, fully dynamic, batch updates, spanning tree

ACM Reference Format:
Anonymous Author(s). 2025. Fully Dynamic Rooted Spanning Tree on GPU. In *Proceedings of International Conference on Parallel Processing (ICPP'25)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The efficient processing of large graphs is increasingly vital across various computational domains, as recent studies underscore the growing complexity and scale of networks [20, 21]. The rise of online social networks and other dynamic environments demands rapid computations of various graph metrics [13, 25]. Real-world networks, such as those in transportation, biology, and social platforms, experience continuous changes or "*churn*". For instance, on a popular social media platform "*Reddit*", users frequently join and leave *subreddits*, leading to constant additions and deletions of connections within the underlying network. These frequent updates pose a significant challenge: efficiently updating vital graph metrics as the graph undergoes changes.

Traditional methods of analysing static snapshots of these networks fall short, given the rapid pace of these changes. This problem has created a new demand for the development of algorithms tailored for dynamic graphs, where edges can be frequently inserted or deleted. A common approach is to accumulate updates into batches that can be processed in parallel. This approach improves performance as it reduces redundant computations and increases parallelism, although it introduces its own set of challenges.
Spanning trees are fundamental to many graph-related applications, playing a critical role in fields like telecommunications, the Internet of Things, transportation, and more. Many crucial network tasks, such as database maintenance, can be executed efficiently using a tree that spans the network. Maintaining spanning trees efficiently in dynamic networks is more practical and resource-efficient than rebuilding them from scratch after each update. This approach not only accelerates tree construction, but also significantly improves the performance of dependent dynamic algorithms such as 2-connected components (BCC), minimum weight spanning Tree (MST), centrality measures, and others.

For example, updating a rooted spanning tree is required as a subroutine in dynamic batch updates of MST [25] and bi-connected components BCC [10] algorithms. However, the current MST algorithm resorts to completely rebuilding the tree, even when updates to an existing rooted spanning tree would suffice. In the dynamic BCC algorithms, the dynamic maintenance of a rooted spanning tree is required. While some level of parallelism is achieved by processing multiple broken trees concurrently, but the path reversal within each tree is done serially by a single thread. Although this coarse-grained parallelism suits multicore CPU, it is not ideal for modern GPUs, where fine-grain parallelism is expected. Additionally, the unique memory hierarchy of GPUs, coupled with the irregular data access patterns in graphs, necessitates careful design of data layouts and memory accesses. Thus, designing graph algorithms for GPUs requires meticulous attention to both parallelism and memory management.

In this paper, we propose GPU based parallel algorithms to dynamically update a rooted spanning tree or forest following the insertion or deletion of a batch of edges. We now formally define the problem.

Dynamic Rooted Spanning Forest (D-RSF)

Input: An undirected graph $G = (V, E)$, a rooted spanning forest F of G, a batch B of edges, and an operation op $\in$ {insert, delete}.
Output: A rooted spanning forest F' of the updated graph G', where

$$G' = \begin{cases} G + B & \text{if op = insert,} \\ G - B & \text{if op = delete.} \end{cases}$$

Given a graph G with V vertices and E edges, and a rooted spanning forest F of G, and a batch B of edges to be inserted or deleted according to an operation **op** $\in$ {insert, delete} the goal is to efficiently maintain F through these dynamic updates. Specifically, the aim is to update the spanning forest dynamically from the current solution, rather than reconstructing the spanning forest from scratch after each operation.

ICPP'25, San Diego, CA
2025. ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Related Work

Over the decades, spanning trees have been widely studied across various computational models, with recent focus shifting towards dynamic spanning trees due to the need for efficient algorithms capable of handling dynamic updates.

Static Algorithms. Harish et al. [8] pioneered BFS implementations on modern GPUs, later extended by Merrill et al. [16], who proposed a work-efficient BFS algorithm using the Prefix Sum technique. While BFS-based methods are efficient for graphs with small diameters, they suffer from performance degradation as the graph depth increases [27].

Beyond BFS, different parallel algorithms for constructing spanning trees have also been explored. Cong et al. [4] proposed a multicore algorithm based on the parallel Union-Find technique, combining grafting and broadcasting to reverse paths and construct a rooted spanning tree. However, the GPU implementation of PR-RST algorithm experiences computational overhead due to multiple rounds of pointer jumping, leading to increased complexity and irregular memory access patterns.

However, the key limitation of all the above mentioned static algorithms is that they do not utilise the edges in the existing spanning forest, leading to redundant computations, even for small updates (e.g., 1-2%), requiring a full run on the entire graph.

Sequential Dynamic Algorithms. To address this issue, Eppstein et al. proposed the *Sparsification Technique*, a widely recognized paradigm in sequential settings for designing dynamic graph algorithms [6], applicable to connected components, biconnected components, and minimum weight spanning trees. Nevertheless, the sparsification technique has notable limitations, including its inherent sequential nature and its focus on single-edge updates. Although this approach has been successfully adapted for dynamic connected components [24], it proved unsuitable for maintaining minimum weight spanning trees in multicore architectures [25], likely due to associated overheads. Furthermore, empirical studies by Haryan et al. [10] demonstrated that the sparsification technique falls short in practice compared to parallel static algorithms for maintaining biconnected components. The exploration of the applicability of the sparsification paradigm to GPU architectures still remains limited, presenting an open research avenue.

Parallel Dynamic Algorithms. To overcome the limitations of sequential dynamic algorithms, parallel batch dynamic algorithms have been explored for maintaining spanning forests under batch updates. An algorithm is referred to as *fully dynamic* if it handles both edge insertions and deletions, whereas *incremental* and *decremental* algorithms support only insertions or deletions, respectively. To the best of our knowledge, three notable contributions on fully dynamic spanning forests are [1, 5, 15].

The HDT (Holm, de Lichtenberg, and Thorup) algorithm [1] and the cluster forest algorithm [5] provide theoretical guarantees on algorithmic depth, but their practical adaptation to specific architectures such as multi-core processors or GPUs remains unexplored. In particular, adapting these algorithms to GPU architectures is non-trivial and remains an open research challenge. A key limitation of the HDT algorithm is its requirement to maintain $O(\log V)$ spanning forests within a hierarchical data structure, leading to a space complexity of $O(V \log V + E)$.

The cluster forest algorithm [5] performs two independent graph traversals in parallel from the endpoints of each deleted edge. While this kind of parallelism is effective on multi-core architectures, it is not well-aligned with the parallel execution model of GPUs due to their architectural constraints. Similarly, the parallel algorithm proposed in [15], though implemented for multi-core systems, lacks theoretical guarantees on depth. This algorithm is based on a static parallel BFS, with its worst-case depth bounded by $O(V + E)$, where V and E denote the number of vertices and edges in the graph.

In summary, existing approaches either prioritize theoretical guarantees or focus on architecture-specific optimizations, but not both. In contrast, our work aims to bridge this gap by designing algorithms that offer theoretical guarantees while being well-suited for practical deployment on GPU architectures. A comprehensive comparison of existing works with our proposed approach is presented in **Table 1**.

Table 1: Depth comparison of algorithms for batch edge operations in dynamic graph algorithms, with SG-ET and HS-ET representing our proposed contributions.

Algorithms	Batch-Insert	Batch-Delete	Space
HDT [1]	$O(\log V)$	$O(\log^3 V)$	$O(V \log V + E)$
CF [5]	$O(\log^3 V)$	$O(\log^3 V)$	$O(V + E)$
SG-ET (ours)	$\boldsymbol{O(\log V + \log^2 k)}$	$\boldsymbol{O(\log V + \log^2 k)}$	$\boldsymbol{O(V + E)}$
HS-ET (ours)	$\boldsymbol{O(\log^2 V)}$	$\boldsymbol{O(\log^2 V)}$	$\boldsymbol{O(V + E)}$

Past work on Dynamic Trees. Dynamic spanning tree problems and dynamic trees are related, but they have different goals. Dynamic trees specifically deal with maintaining a forest of trees, and support insert and delete operations on edges, while ensuring the structure remains **acyclic**. In contrast, dynamic spanning tree problems operate on general graphs, maintaining a spanning tree while accommodating edge insertions and deletions, making them inherently more complex.

The concept of dynamic trees was pioneered by Sleator and Tarjan [22] (1981) with the introduction of link/cut trees for fast network flow applications, later adapted by Frederickson [7] (1983) for dynamic MST updates. Following that subsequent researches have expanded this area extensively in both sequential and parallel settings [11, 26].

1.2 Our Contributions

Designing parallel dynamic algorithms for D-RSF, where the algorithm's depth depends on the number of affected key edges (m) or vertices (n), poses significant challenges. The algorithms presented in this paper directly address these challenges. Our design focuses on achieving low-depth parallel algorithms to ensure practical efficiency, particularly on GPU architectures. To the best of our knowledge, there are no existing GPU-based parallel algorithms for maintaining rooted spanning forests under dynamic updates. Our key contributions are summarized below:

- We design and implement four fully dynamic parallel algorithms for the D-RSF problem, specifically tailored for GPU architectures. These algorithms provide theoretical guarantees while demonstrating strong practical performance.

- We design a GPU-based parallel subroutine to reverse multiple paths in a rooted spanning forest, integrating ideas from Hooking-Shortcutting, Broadcasting and Euler-Tour techniques to solve the D-RSF problem.
- We provide a formal proof of correctness for our generic algorithm to solve the D-RSF problem.
- Our GPU-based parallel algorithms significantly outperform existing state-of-the-art static parallel GPU algorithms, achieving speedups of up to 500× for deletions and up to 900× for insertions. We also identify and analyze key factors that impact the performance of our algorithms.
- Beyond maintaining rooted spanning forests, our proposed algorithms can also be applied to maintain the connected components of a graph under dynamic updates, including both edge insertions and deletions.

2 Generic Algorithm for D-RSF

In this section, we begin by introducing the necessary notations that are used throughout the description of our D-RSF algorithm. We then present an overview of the algorithm, followed by two important subroutines to address the subproblems of *oriented replacement edges* and *path reversal*. Finally, we describe the complete generic algorithm using the introduced notation and subroutines.

2.1 Notation

Let $G = (V(G), E(G))$ be an input undirected graph, where $V(G)$ and $E(G)$ denote the set of vertices and edges, respectively in G. A forest $F = (V(F), E(F))$ of G is said to be a *rooted spanning forest* of G if $V(F) = V(G)$ and for every vertex v in F, there exists a parent except for one vertex in each component. The vertices that do not have parents correspond to *root vertices*. For an edge $e = (x, y)$ in a rooted tree, where y is the parent of x, we say that the *orientation* of e is from x to y ($x \longrightarrow y$).

Table 2: Notations

Notation	Description
G	An undirected graph
$V(G), n$	The number of vertices in G
E	The number of edges in G
$E(G)$	The set of edges in G
F	A rooted spanning forest of G
$x \longrightarrow y$	Orientation of an edge from child x to its parent y in F
m	The number of key edges
k	The number of components (trees) in F
G'	Graph G after a batch update
F'	A rooted spanning forest of G'
$\hat{F}$	$\hat{F} = F - B$ for deletion operation; $\hat{F} = F$ for insertion operation

Let F be a rooted spanning forest of G on n vertices. The resultant graph obtained after inserting or deleting a batch B of edges in G is denoted by G'. We use M to denote a set of potential edges that help to establish the connectivity across the trees in a spanning forest of G' and such edges are referred as **key edges**. In case of delete operation, $M = E(G) - E(F) - B$, where as $M = B$ for insert operation. We use m to denote the number of key edges.

For delete operation, $\hat{F}$ denote the forest obtained after deleting edges of B from F, whereas $\hat{F}$ remains same as F for insert operation. Each tree T_i in $\hat{F}$ is uniquely identified by its root vertex r_i. For each vertex v in $\hat{F}$, the root vertex of the tree containing v is called as *representative*, which is denoted by $rep[v]$. An edge $(u, v) \in M$ is a *cross edge* if $rep[u] \neq rep[v]$. Let E_{cross} be the set of all cross edges in G' with respect to $\hat{F}$. An edge set $E_r \subseteq E_{cross}$ is referred to as *replacement edges* if $|E_r| = k - \ell$, and at most one edge from E_r appears between any two trees in $|\hat{F}|$, where k and ℓ denotes the number of components in $\hat{F}$ and G', respectively.

2.2 Overview

We now present a high-level overview of our generic parallel algorithm to solve the D-RSF problem. The dynamic algorithm updates the rooted spanning forest of a graph in parallel, based on a batch of edges to be inserted or deleted, without reconstructing it from scratch.

First, depending on the type of update operation, we either insert the given batch of edges into the underlying graph or remove them from both the forest and the underlying graph. Next, we identify *replacement edges* from the available set of *key edges* to reestablish connectivity across the trees in the forest. Once the replacement edges are identified, assigning their orientations arbitrarily can lead to conflicts where a vertex may end up with multiple parents. For example, in Figure 1(b), both vertices 6 and 17 could attempt to become the parent of vertex 10 if orientations are assigned arbitrarily. Such conflicts can result in the loss of valid orientations for certain edges. Therefore, it is necessary to systematically determine the *orientations* of the replacement edges before including them in the forest. The task of finding *oriented replacement edges* constitutes our first subproblem.

Although it is possible to assign orientations to replacement edges without causing internal conflicts, these new orientations may still overwrite the parent relationships of existing tree edges. For instance, in Figure 1(b), the addition of a replacement edge $(6, 10)$ could reassign the parent of vertex 6 to vertex 10, thereby causing the loss of the original tree edge $(4, 6)$.

To overcome this, we introduce a second subproblem, which involves reversing the directions of a few tree edges to maintain a valid rooted forest. More precisely, we reverse the orientations along certain paths (e.g., the path from vertex 6 to 1 in Figure 1 (b)), and then later insert the replacement edges to obtain the new forest. Overall, the D-RSF problem is divided into two sub-problems. The first sub-problem involves finding the **orientated replacement edges** across the disconnected trees, while the second aims to reverse the orientations of multiple paths in a forest, shortly called **reverse paths**.

2.3 Subroutines

To address the two sub-problems proposed earlier, we now present the necessary subroutines. We defer the algorithm details required for these subroutines to later subsections.

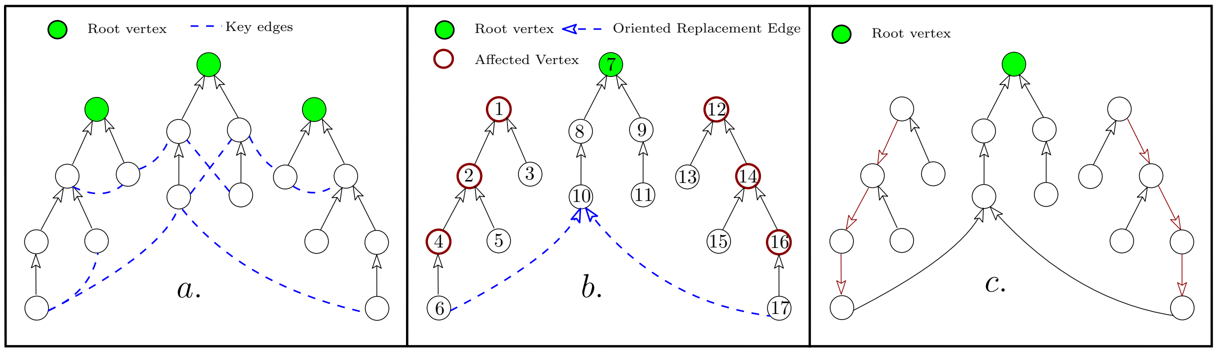


Figure 1: Overview of the D-RSF algorithm. (a) The black solid lines represent the rooted spanning forest, while the blue edges denote the key edges. (b) Replacement edges are selected from the key edges and oriented appropriately. (c) Parents are updated for affected vertices during path reversal to obtain an updated rooted spanning forest.

FindOrientedReplEdges($rep[\cdot]$, M): Given the representative array $rep[\cdot]$ for a rooted forest $\hat{F}$ and a set M of key edges, this subroutine identifies oriented replacement edges from M to establish connectivity across the trees in $\hat{F}$.

ReversePaths(R, $parent[\cdot]$)): Given a rooted forest $\hat{F}$ (represented using the $parent[\cdot]$) and a set R of pairs, where each pair denotes the end vertices of a path in $\hat{F}$. This subroutine reverses the parent-child relationships of all vertices along the paths in R in parallel.

Algorithm 1: A GPU Algorithm for Dynamic Rooted Spanning Forest

```
Input: A Graph $G = (V, E)$, a rooted spanning forest $F$ of $G$
       represented by $parent[\cdot]$, a batch of edges $B$, and an
       operation op $\in$ {insert, delete}.
Output: A Rooted Spanning Forest $F'$ of $G' = G$ op $B$
1  $M = B$
2  if operation is delete then
3      $M = (E(G) - E(F)) - B$
4      for each edge $(u, v) \in B$ in parallel do
           /* Let $v$ be the parent of $u$ */
5          $parent[u] \leftarrow u$
6  $rep[\cdot] \leftarrow$ parallel pointer jumping on $parent[\cdot]$
7  $E' \leftarrow$ FindOrientedReplEdges($M$, $rep[\cdot]$)
8  for each oriented edge $(u, v) \in E'$ in parallel do
       /* Let $v$ be the parent of $u$ */
9      $R = R \cup \{(u, rep[u])\}$
10 $F' \leftarrow$ ReversePaths($\hat{F}$, $R$, $parent$)
11 Update $parent[\cdot]$ in parallel for each edge in $E'$
12 Insert/Delete $B$ of edges to/from $G$
```

2.4 Generic Algorithm

An instance of the D-RSF problem involves an undirected graph G, a rooted spanning forest F of G, and a batch B of edges for insertion or deletion. Algorithm 1 begins by identifying the operation type. For deletions, edges in B are removed from the non-tree edges, and the key edges M are computed in parallel (Line 3). Subsequently, for each tree edge in B, the forest F is updated in parallel by setting the parent node to itself (Line 5), effectively deleting all the tree edges in B from the forest. For insert operation, since no non-tree edge exists between disconnected trees in the forest, B itself becomes key edges in Line 1. After updating the forest and obtaining key edges, the algorithm proceeds to determine the representative of each vertex, resulting in the $rep[\cdot]$, by applying pointer jumping on the parent array (Line 6). This ensures that each vertex in the forest can retrieve its root vertex quickly in the subsequent tasks. The algorithm then identifies oriented replacement edges E' from the set M of key edges using the parallel subroutine FindOrientedReplEdges (Line 7). As illustrated in Figure 1, when including an edge (6, 10) and setting its orientation such that 10 is a parent of 6, it may be necessary to reverse the orientations of the edges involved in the path from 6 to 1. Therefore, before incorporating the oriented replacement edges into the forest, we first identify the end vertices of the paths whose orientations must be reversed (Line 9). The paths represented by pairs in R are then reversed using the parallel subroutine ReversePaths (Line 10). Finally, the new replacement edges are incorporated into the updated forest F' (Line 11) using $|E'|$ threads, ensuring that the spanning forest remains properly rooted after edge updates.

3 Replacement Edges and Path Reversal

The two main subproblems in our generic algorithm to solve D-RSF are finding oriented replacement edges and reversing multiple independent paths, both in parallel. We propose Supergraph (SG) and Hooking Shortcutting (HS) techniques to address the first problem. Further, we describe Broadcasting (BC) [4] and design Euler Tour (ET) based technique to solve the second. In a nutshell, we propose the following four algorithms to solve D-RSF, each corresponding to a different combination of the techniques.

(1) SG-ET (Supergraph with Euler Tour)
(2) SG-BC (Supergraph with Broadcasting)
(3) HS-ET (Hooking Shortcutting with Euler Tour)
(4) HS-BC (Hooking Shortcutting with Broadcasting)

3.1 Supergraph for Oriented Replacement Edges

Given a forest $\hat{F}$, a representative array $rep[\cdot]$ to hold representative for every vertex in $\hat{F}$ and a set M of key edges, we now identify oriented replacement edges. In this supergraph (SG) approach we construct a supergraph $\tilde{G}$ and a hash table using the forest and key edges. The supergraph helps to maintain the topological structure across the trees in the forest, whereas the hash table is useful to retrieve a key edge corresponding to any edge in the supergraph. We first construct a supergraph $\tilde{G}$, in which each vertex corresponds to a tree in the forest $\hat{F}$ and an edge in the supergraph corresponds to a cross edge across the trees in the forest. In particular, for each root vertex u in $\hat{F}$ there is a vertex in $\tilde{G}$. We go through all key edges $e = (u, v)$ in parallel and the edge (u, v) is marked as *cross edge* if $rep[u] \neq rep[v]$. We now use a temporary array *sedges* to capture the potential edges to be added in the supergraph. For each cross edge $e_i = (u, v)$ in parallel, we include an edge in $\tilde{G}$ by assigning $sedges[i] = (rep[u], rep[v])$, and insert a key-value pair in a hash table H, where $key = (rep[u], rep[v])$, and $value = (u, v)$. At this stage, the number of edges in the supergraph equals to the number of cross edges. Multiple cross edges over the same two trees result in duplicate edges in the supergraph. To eliminate such redundant edges, we apply parallel sorting and parallel compaction on *sedges*. Although multiple pairs with same key attempt to write in the hash table in parallel, eventually no two pairs in the hash table have same key, due to race conditions. In other words, for each edge in the supergraph $\tilde{G}$, we maintain a unique cross edge in the hash table.

We now apply PR-RST algorithm [4] on the super graph $\tilde{G}$ to obtain a rooted spanning forest $\tilde{F}$ of $\tilde{G}$. By treating the edges of $\tilde{F}$ as keys, we can retrieve the corresponding cross edges from the hash table and these cross edges become oriented replacement edges. For each edge oriented (u', v') in $\tilde{F}$, where u' is the parent of v', we retrieve a oriented cross edge (u, v) from the hash table H, where u is the parent of v.

Lemma 3.1. *Given a forest $\hat{F}$ with k trees and m key edges, oriented replacement edges can be obtained using the supergraph approach in $O(\log m + \log^2 k)$ depth and $O(m + k \log k)$ work in the worst case.*

Proof. Identifying the non-cross edges among the m key edges is achieved in constant time, requiring $O(m)$ total work. These edges are subsequently removed using parallel compaction [12], which operates in $O(\log m)$ depth and $O(m)$ work. The remaining cross edges are then inserted in supergraph $\tilde{G}$ and parallel hash table H in $O(1)$ time, maintaining a total work complexity of $O(m)$. Duplicate edges are then eliminated in $\tilde{G}$, through parallel sorting [19] and parallel compaction, which requires $O(\log m)$ depth and $O(m)$ work. Combining these steps, eventually the auxiliary graph $\tilde{G}$ is constructed in $O(\log m)$ depth with $O(m)$ total work.

To compute the rooted spanning forest $\tilde{F}$ of $\tilde{G}$, *PR-RST* algorithm [4] takes $O(\log^2 k)$ depth and $O(k \log k)$ work for a graph with k vertices.

For each edge in $\tilde{F}$, the corresponding edge in G can be retrieved in parallel in constant time using the hash table H, resulting in $O(k)$ total work. By combining all the steps, the oriented replacement edges are obtained in $O(\log m + \log^2 k)$ depth performing $O(m + k \log k)$ work. □

The merit of this approach is that the depth of this algorithm is independent on the number of vertices n, and only depends on the number of disconnected components and the key edges. However, m can be large in the case of deletion operation, and thus we aim to design a parallel algorithm that minimises the dependency on m in the next subsection.

3.2 Hooking and Shortcutting for Oriented Replacement Edges

Algorithm 2: Oriented Replacement Edges using Hooking and Shortcutting

```
Input: key edges $M[\cdot]$, and representative array $rep[\cdot]$
Output: At most $k - 1$ oriented replacement edges
1 $replEdges \leftarrow$ Hooking-Shortcutting$(M, rep)$
2 for each $e_i = (u, v) \in replEdges$ in parallel do
3     $sfEdges[i] = (rep[u], rep[v])$
4     $H$.insert$(\langle rep[u], rep[v] \rangle, \langle u, v \rangle)$
5 $O \leftarrow$ Eulerian-Tour$(sfEdges)$
6 $R \leftarrow H$.retrieve$(O)$
```

The hooking and shortcutting (HS) based parallel algorithm is a well-known technique to obtain a spanning forest [23]. We adapt this subroutine to retrieve a set of replacement edges and then systematically orient them.

Given the set of key edges $M[\cdot]$ and the representative array $rep[\cdot]$ for a rooted forest $\hat{F}$, our method proceeds in two phases. In **Phase 1 (Line 1)**, we apply the Hooking-Shortcutting procedure to identify at most $k - 1$ replacement edges that can reconnect the disconnected components.

In **Phase 2 (Line 2 - Line 6)**, we orient the selected replacement edges. To maintain the topological structure across the trees, we first construct an auxiliary forest by storing, for each replacement edge (u, v), a key-value mapping in a hash table H, where the key is $\langle rep[u], rep[v] \rangle$ and the value is $\langle u, v \rangle$ (Line 2). This auxiliary forest, formed by the keys in H, represents a super-forest over the component representatives. We then apply the Eulerian Tour algorithm to this super-forest (Line 5) to obtain a rooted structure. Finally, we retrieve the corresponding original edges from the hash table H (Line 6) to generate the final set of *oriented replacement edges*. For hash table implementation, we adapted an existing open-source code available on GitHub [17], to suit our use case.

Lemma 3.2. *Algorithm 2 identifies oriented replacement edges in a graph using Hooking and Shortcutting operations in $O(\log^2 n)$ depth and $O((m + n) \log n)$ work.*

Proof. The depth of the HS approach is dominated by Algorithm 2, which requires $O(\log n)$ iterations in the worst case, resulting in a depth of $O(\log^2 n)$ and a work complexity of $O((m + n) \log n)$ [23].

The loop in Line 2 takes constant depth and $O(k)$ work. The trees in the auxiliary forest are then rooted using a parallel Eulerian tour algorithm (Line 5), which operates in $O(\log k)$ depth and performs $O(k)$ work [18]. Finally, all the oriented replacement edges can be retrieved in constant time from the hash table.

Combining these steps, the total depth of the algorithm is bounded by $O(\log^2 n)$, with a total work complexity of $O((m+n)\log n)$. □

A key merit of this approach is that its depth depends solely on the number of vertices, not the number of edges. While the HS technique is primarily used to identify oriented replacement edges, it serves as a parallel static algorithm for obtaining a rooted spanning tree.

3.3 Broadcasting Based Path Reversal

The purpose of this subroutine is to reverse the orientation of all disjoint paths of a forest, which are specified by R, where each path is represented by a pair (u_i, r_i). We use the broadcasting method by Cong et al. [4] to efficiently reverse multiple paths in a forest in parallel. The algorithm begins by identifying all the *on-path* vertices that lie on the path from u_i to r_i for each pair in R simultaneously. Once these vertices are identified, the algorithm reverses their parent-child relationships in parallel i.e., if $u = parent[v]$, we now set $parent[u] = v$.

For each vertex u in the tree, an array of size $O(\log n)$, referred to as the *special ancestor*, is maintained. This array stores all ancestors of u at distances equal to powers of 2, using the pointer jumping process. In the subsequent $\log n$ iterations, all vertices that lie on the path from u_i to r_i can be identified in parallel using special ancestors. For further details, we refer to [4].

LEMMA 3.3. (Lemma 1 and 2 from [4]) *Given a spanning forest $\hat{F}$ and a set R of pairs, where each pair represents the end vertices of a disjoint path in $\hat{F}$, the paths represented by R can be reversed using the broadcasting technique in $O(\log n)$ depth and $O(n \log n)$ work.*

The primary drawback of this technique is its $O(n \log n)$ space complexity. Additionally, it faces significant limitations when implemented on a GPU. The multi-step pointer jumping required to construct the special ancestor array involves multiple iterations and external synchronization, resulting in considerable runtime overhead. To address these inefficiencies, we propose an alternative method in the next section that consumes linear space and is better suited for the GPU architecture.

3.4 Eulerian Tour Based Path Reversal

Given a rooted forest $\hat{F}$ and a set R of pairs representing the end vertices of paths in $\hat{F}$ whose orientation needs to be reversed, we design Algorithm 3 to achieve this using *Euler Tour*.

In $\hat{F}$, let $firstEdge[v]$ and $lastEdge[v]$ denote the first and last edges incident on each vertex v. Each tree in $\hat{F}$ has its own Eulerian tour, where each tree edge appears twice.

As shown in Line 1 of Algorithm 3, we first compute the Euler tours for all trees in $\hat{F}$ and assign ranks to the edges using the parallel algorithm by Polák et al. [18]. Based on these ranks, we assign *start* and *finish* times to each vertex, categorizing them into three groups as outlined in Lines 2–10. The assigned *start* and *finish* times for all vertices are shown in Figure 2.

For each path (x_i, r_i), we update $support[r_i]$ with x_i (Line 12), enabling constant-time retrieval in subsequent steps. In the last for loop of Algorithm 3, we go through all edges and identify the affected edges in Line 15, and reverse their orientations. Each vertex u knows its representative rep$[u]$ (the tree to which it belongs). Using this information, u can easily determine the end vertex of the path (remember *support* was updated earlier (Line 12)). Thus, v can efficiently verify whether it lies on the path.

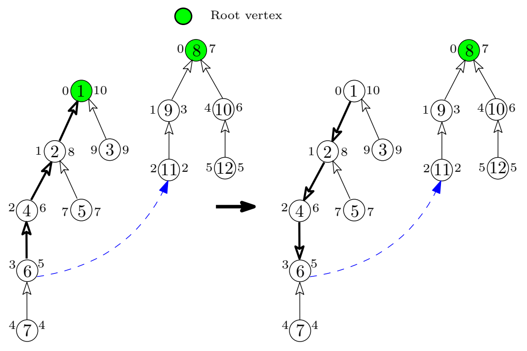


Figure 2: Orientation of affected edges, show in solid thick, is reversed. Numbers shown on left and right sides of each vertex correspond to start and finish times, obtained using Euler-Tour.

```
Algorithm 3: ReversePathsEulerTour
   Input: A rooted spanning forest $\hat{F}$, set of vertex pairs $R$,
          $rep[\cdot]$ and $parent[\cdot]$ arrays
   Output: Updated $parent[\cdot]$ in the Forest $\hat{F}$
 1 $rank[\cdot] \leftarrow$ EulerianTour($parent$)
 2 for each vertex $v$ in parallel do
 3     if $v$ is a Leaf Vertex then
 4         $start[v] = finish[v] = rank[\langle v, parent[v]\rangle]$
 5     else if $v$ is a Root Vertex then
 6         $start[v] = rank[firstEdge[v]]$
 7         $finish[v] = rank[lastEdge[v]] + 1$
 8     else if $v$ is an Intermediate Vertex then
 9         $start[v] = rank[firstEdge[v]]$
10         $finish[v] = rank[\langle v, parent[v]\rangle]$
11 for each vertex pairs $(x_i, r_i) \in R$ in parallel do
12     $support[r_i] = x_i$
13 for each edge $(u, parent[u])$ in $\hat{F}$ in parallel do
14     $r = rep[u], x = support[r]$
15     if $(start[r] < start[u] \leq start[x])$ and
        $(finish[r] > finish[u] \geq finish[x])$ then
16         $v = parent[u]$
17         $parent[v] = u$
```

THEOREM 3.4. *Given a spanning forest $\hat{F}$ and a set R of pairs, where each pair represents the end vertices of a disjoint path in $\hat{F}$, Algorithm 3 reverses the paths represented by R in $O(\log n)$ depth and $O(n)$ work.*

PROOF. The depth of the algorithm is primarily determined by Line 1. The *rank* array is computed in $O(\log n)$ depth, performing

$O(n)$ work, as established by Polák et al. [18]. Using the *rank* array, the *start* and *finish* times for all vertices in $\hat{F}$ (Lines 2–10) are calculated in $O(1)$ depth with n threads, resulting in $O(n)$ work.

Line 12 executes in $O(1)$ depth using k threads, performing $O(k)$ work to update the support information. Subsequently, all paths are reversed (Line 13) in $O(1)$ depth by swapping parent-child relation in parallel, using n threads and performing $O(n)$ work.

Since each vertex in the forest is processed exactly once and no vertex is revisited, the total work across all operations remains $O(n)$. Thus, Algorithm 3 completes the reversal of all paths in the forest in $O(\log n)$ depth and $O(n)$ work. □

4 Correctness and Complexity Analysis

In this section, we analyze the depth and work complexity of the four proposed algorithms and present a proof of correctness for the generic algorithm (Algorithm 1). We then separately analyze the depth complexity for the insertion and deletion operations.

Theorem 4.1. *The* SG-BC *and* SG-ET *algorithms* ***insert*** *a batch B of edges in $O(\log n + \log^2 k)$ depth, while the* HS-BC *and* HS-ET *algorithms require a depth of $O(\log^2 n)$. The total work performed by all algorithms is $O(n \log n)$.*

Proof. The depth of Algorithm 1 for insertion is primarily determined by Line 7 and Line 10. The FindOrientedReplEdges subroutine contributes differently depending on the approach: the SG-based method has a depth of $O(\log m + \log^2 k)$ and performs $O(m + k \log k)$ work (Lemma 3.1), whereas the HS-based method requires a depth of $O(\log^2 n)$ and performs $O((m + n) \log n)$ work (Lemma 3.2). The PathReversal subroutine contributes a depth of $O(\log n)$ and performs $O(n \log n)$ work for the Broadcasting approach (Lemma 3.3), while the EulerianTour approach contributes $O(n)$ work (Theorem 3.4).

Consequently, the overall depth for the SG-BC and SG-ET algorithms is upper bounded by $O(\log m) + O(\log n) + O(\log^2 k)$, performing $O(m + n \log n)$ work in the worst case (WC). For the HS-BC and HS-ET algorithms, the depth is $O(\log^2 n)$, and the work complexity is $O((m + n) \log n)$ in the worst case.

For the insert operation, m corresponds to the number of edges in the batch, as indicated in Line 1 of Algo. 1. In batch dynamic algorithms, the batch size B is typically bounded by $O(n)$ [1]. Consequently, the $O(\log m)$ term in the depth of the SG-BC and SG-ET algorithms simplifies to $O(\log n)$. Hence, the overall depth for the SG-BC and SG-ET algorithms becomes $O(\log n + \log^2 k)$, while the work complexity for all the algorithms is bounded by $O(n \log n)$. □

Theorem 4.2. *The* SG-BC *and* SG-ET *algorithms* ***delete*** *a batch B of edge in $O(\log n + \log^2 k)$ depth, while the* HS-BC *and* HS-ET *algorithms require a depth of $O(\log^2 n)$. Both algorithms perform total work of $O(E + n \log n)$, where n denotes the number of vertices and E denotes the number of edges in G.*

Proof. The depth of Algorithm 1 for the deletion operation incurs additional overheads compared to the insertion, because the number of key edges in delete operation is at most E. Also, updating the data structure in Line 2 and computing the key edges in Line 3 introduce overheads due to parallel compaction, which is bounded by $O(\log E)$ depth and $O(E)$ work. In the worst case, $E = O(n^2)$, leading to a maximum depth of $O(\log n^2)$ which simplifies to $O(\log n)$. When combined with other terms, this introduces an additional overhead of $O(\log n)$ depth for the deletion operation; however, this additional overhead does not affect the asymptotic depth of the algorithms. □

The space complexity of the Algorithm 1 is bounded by $O(V + E)$, where V represents the number of vertices and E the number of edges in the graph. This complexity arises from storing the graph (using an edge list) and auxiliary structures such as the parent array, representative (rep) array, and similar data structures. Temporary storage for operations such as compaction, sorting or hash table also scales with V or E. Additionally, $O(B)$ space is required for handling a batch B of edges.

Further, broadcasting approach requires $O(V \log V)$ space as each vertex stores upto $\log V$ ancestors [4]. Consequently, the SG-BC and HS-BC algorithms consume $O(V \log V + E)$ space, whereas SG-ET and HS-ET algorithms consume $O(V + E)$ space.

We now proceed to prove the correctness of Algorithm 1 and Algorithm 3.

Lemma 4.3. *Let R be a set containing k pairs of vertices (x_1, y_1), $(x_2, y_2), \dots, (x_k, y_k)$ in a forest $F = \{T_1, T_2, \dots, T_k\}$, where y_i is the root of a tree $T_i \in F$ and x_i is a vertex in T_i. Using the start and end times obtained from the Eulerian tour, Algorithm 3 correctly reverses all paths from each x_i to its corresponding root y_i.*

Proof. Consider a tree T_i and vertex pair (x_i, y_i). Let U to be the set of vertices u_i such that $first[x_i] < first[u_i] \le first[y_i]$ and V to be the set of vertices v_i such that $last[x_i] > last[v_i] \ge last[y_i]$. Then $Path(x_i, y_i) = \{w_i \mid w_i \in U \cap V\}$ consists of all vertices in the simple path from x_i to y_i. $\forall\, i, j \in [1, k]$ such that $i \neq j$, since any two trees T_i, T_j are disjoint, it follows that $Path(x_i, y_i) \cap Path(x_j, y_j) = \emptyset$. Thus, all paths can be reversed without conflicts. □

Theorem 4.4. *Given an undirected graph G, a rooted spanning forest F of G, a batch of edges B to insert or delete, Algorithm 1 produces a valid rooted spanning forest F' of $G \pm B$.*

Proof. Let G' be the graph and $\hat{F}$ the forest obtained after the batch updates. Let ℓ denote the number of connected components in G' and $k = |\hat{F}|$ represent the number of trees in $\hat{F}$. Using the function OrientedReplEdges (Line 7), we compute the set of oriented replacement edges $E'_r = \{e'_1, e'_2, \dots, e'_{k-\ell}\}$.

Let $e_k = (x, y)$ be a *replacement edge* in E'_r. Consider two trees T_i and $T_j \in \hat{F}$ rooted at r_i and r_j, respectively, where $x \in T_i$ and $y \in T_j$. The simple path from x to the root r_i is denoted as $P = \{x, v_1, v_2, \dots, r_i\}$. After applying ReversePaths (Line 10), the parent-child relationships along P are reversed, resulting in every vertex $v \in T_i$ having a parent, except for x. When the edge (x, y) is inserted, x acquires y as its parent in Line 11.

The addition of each *replacement edge* reduces the number of root vertices in $\hat{F}$ by one. Starting with k roots, the insertion of $k - \ell$ replacement edges decreases the root count to $k - (k - \ell) = \ell$. Therefore, the resulting forest $\hat{F}$ is a valid rooted spanning forest of G' with exactly ℓ trees, one per connected component of G'. □

[1]For scenarios involving a significantly large number of updates, it may be computationally more efficient to recompute the solution from scratch rather than incremental updates.

5 Implementation

To improve GPU utilization and take into account the massive thread-level parallelism available in CUDA, we follow several implementation strategies, which are described below.

Update Data-Structure:

Each edge is represented as a 64-bit integer in GPU memory. To identify key edges in case of deletion (Line 3 in Algorithm 1), for all edges in E, we first check if an edge is a tree edge using the parent array, if not, perform a binary search in the batch B, and mark the relevant edges. Marked edges are removed using *cub::DeviceSelect::Flagged* (stream compaction). For insertion operations, the input batch B is directly added to the graph.

Pointer Jumping: Pointer jumping is utilized in multiple contexts, such as identifying the roots of all trees in a forest, constructing the special ancestor array, etc. To optimize performance, we execute five pointer jumps per thread in parallel before applying global synchronization. This approach empirically proves to be faster than synchronizing after each pointer jump, as it reduces the frequency of costly global synchronizations on GPUs.

Supergraph Creation:

In Section 3.1, every root vertex of a tree in the spanning forest becomes a vertex in the supergraph $\tilde{G}$. The vertices in the supergraph are not necessarily consecutive; however, we make them contiguous to enable direct array indexing, reduce memory usage, and improve graph traversal speed. To achieve this, we use the *cub::DeviceSelect::Flagged* function, to transform non-consecutive vertices into consecutive ones.

Hash Table: Our implementation builds upon an open-source hash table framework available on GitHub [17], which we adapted and optimized for our specific use case. Each edge (u, v) is compactly encoded as a 64-bit key using bitwise shift operations, with its corresponding transformed edge stored as the associated value.

The hash table uses the 64-bit *Murmur2* hash function, though we recognize it may not be the optimal choice for all scenarios. It uses open addressing with linear probing, which offers better cache performance compared to pointer-based chaining methods. Additionally, we optimized the hash table by ensuring a power-of-two capacity, enabling efficient modulus operations through bitwise AND.

6 Experiments and Results

In this section, we introduce the datasets and the configuration of the experimental platform. We then analyse the performance and scalability of the proposed algorithms compared to the static BFS and PR-RST algorithms. Finally, we highlight key insights and the influencing factors of the RSF algorithm.

6.1 Experimental Setup and Dataset

We conduct our experiments on an NVIDIA A100 GPU with 80 GB of on-board memory, based on the Ampere architecture. The GPU features 6912 CUDA cores, 80 Streaming Multiprocessors (SMs), 2.04 TB/s memory bandwidth, 6 MB of L2 cache, and 128 KB of L1 cache per SM. The GPU is connected to an AMD EPYC 7742 CPU with 64 cores, 4 MB of L1 cache, 32 MB of L2 cache, and 256 MB of L3 cache. The reported GPU times exclude memory transfer between CPU and GPU[2]. The number of threads per block was set to 1024 for all experiments. The complete source code used for the experiments is available at here.

To test the capability of our algorithms under dynamic graph updates, we used 13 datasets that include both real-world and synthetic graphs, as listed in Table 3. Sourced from various publicly available repositories, these datasets represent road networks, web graphs, co-purchasing networks, and social networks [2, 3, 14], and range from 1 million to 1 billion edges. They are commonly used in evaluations of GPU-based graph algorithms [10, 18, 23]. Prior to our experiments, we preprocessed each graph by treating directed edges as undirected and removing multiple edges and self-loops, similar to other works.

Finally, to test the scalability of our algorithms, we check their ability to handle different batch update sizes. We use a wide range of batch update sizes, ranging from 100 to 100,000 edges. The batch edges are selected uniformly at random from the graph.

A delete batch created uniformly at random, likely to have fewer tree edges compared to non-tree edges, and thus we choose (approximately 30 - 40%) of the edges from tree edges and the rest from the non-tree edges, while creating delete batches.

For each graph and each batch size, we generate five independent instances. Each instance is executed five times independently, totaling 25 individual executions. The speedup is computed by averaging the ratios of baseline execution time to our method's execution time over these 25 runs. While multiple batch sizes were evaluated to analyze performance trends, we report results primarily for batch size 10,000 for consistency and clarity.

Table 3: Statistics of graphs used in the experiments

Datasets	|V|	|E|
Social Networks		
higgs-twitter	456K	14.8M
hollywood-2009	1.1M	56.3M
com-Orkut	3.07M	117M
soc-LiveJournal1	4.8M	68M
Web Graphs		
uk-2002	18.5M	298M
arabic-2005	22.7M	639M
uk-2005	39.4M	936M
Road Networks		
GAP-road	23.9M	58M
europe_osm	50.9M	54.1M
Kronecker (Synthetic) Graphs		
kron_g500-logn18	262.1K	10.6M
kron_g500-logn19	524K	22M
kron_g500-logn20	1.0M	45M
kron_g500-logn21	2.1M	91M

[2]We assume our algorithms are part of a larger GPU processing pipeline, and thus the input data is already present in the GPU memory.

6.2 Comparison with Static Parallel BFS

In this section, we compare the performance of our algorithms against the static parallel BFS algorithm, which serves as our first baseline.

Parallel BFS algorithms typically assume that the graph is connected; however, when it is not, BFS must be initiated separately for each disconnected component, leading to multiple sequential executions. To eliminate such serial processing, we first add or delete the necessary batch of edges and then identify all disconnected components using the method described in [23]. We subsequently use stream compaction to select all unique representatives and apply parallel BFS starting from these representatives.

Our results show a maximum speedup of 530× for edge deletion (with an average speedup of 160×) and 900× for edge insertion (with an average speedup of 200×), as illustrated in Figures 3 and 4.

The observed improvements in running times can be attributed to the dynamic setting. The size of the largest connected component and the depth of the tree continually change as new batches of edges are inserted or deleted. This variability affects the performance of BFS due to its dependence on the graph's diameter [18]. However, all of our proposed algorithms, being independent of the tree's depth, demonstrated significant speedup compared to the baseline, making them more suitable for real-world dynamic graphs.

6.3 Comparison with static PR-RST algorithm

We now compare our proposed algorithms with the existing multi-core PR-RST algorithm by Cong et al. [4]. While BFS has been extensively studied and widely implemented, the multi-core PR-RST algorithm has not yet seen widespread adoption on GPUs, to the best of our knowledge. To ensure a fair comparison, we implemented the multi-core PR-RST algorithm on GPU. As shown in Figure 5 and Figure 6, our best-performing algorithm achieves, on average, a 13× speedup over PR-RST for deletion operations (with a maximum speedup of 30×) and an 18× speedup for insertion operations (with a maximum speedup of 41×). The primary reason for this significant speedup is that PR-RST, being a static algorithm, does not utilise information from the existing solution and recomputes the rooted spanning forest from scratch, requiring traversal of all edges. Furthermore, the construction of the ancestor array (discussed in Section 3.3) incurs a high number of uncoalesced memory accesses, further degrading performance on the GPU.

Table 4: Running times (in ms) for insertion operation

Filename	BFS	PR-RST	SG-ET	HS-ET	SG-BC	HS-BC
arabic-2005	355.39	1150.89	**36.36**	57.86	66.69	91.46
com-Orkut	71.06	39.02	**10.6**	13.32	13.59	16.91
europe_osm	2318.93	2517.1	**76.31**	154.58	152.26	242.59
GAP-road	894.31	1351.64	**55.95**	62.32	74.74	126.69
higgs-twitter	25.5	38.36	**4.6**	5.81	6.61	6.37
hollywood-2009	730.77	93.99	**7.23**	7.36	9.65	9.3
kron_g500-logn18	716.49	31.73	**4.9**	5.22	8.02	5.81
kron_g500-logn19	1590.79	63.06	**5.55**	6.03	8.59	6.76
kron_g500-logn20	3776.22	106.36	**7.5**	7.05	9.3	8.17
kron_g500-logn21	9497.26	197.58	11.76	**10.57**	13.2	14.91
soc-LiveJournal1	223.46	113.88	**16.25**	16.69	17.9	26.01
uk-2002	4301.79	853.09	**34.6**	50.1	59.21	86.33
uk-2005	**1971.65**	**2149.24**	**53.34**	**108.43**	**118.97**	**228.25**

Table 5: Running times (in ms) for deletion operation

Filename	BFS	PR-RST	SG-ET	HS-ET	SG-BC	HS-BC
arabic-2005	353.92	1149.57	158.91	**88.25**	201.71	128.59
com-Orkut	64.55	51.95	47.22	**34.71**	50.54	36.13
europe_osm	2306.15	2516.99	**88.42**	150.7	188.91	259.07
GAP-road	**884.06**	**1350.66**	**44.35**	**77.8**	**92.95**	**126.67**
higgs-twitter	25.83	38.34	6.31	**4.66**	7.26	4.83
hollywood-2009	790.77	94.04	17.31	**9.25**	18.96	16.35
kron_g500-logn18	716.98	31.68	5.79	**3.24**	6.09	3.84
kron_g500-logn19	1590.79	63.06	8.27	**4.5**	9.38	5.86
kron_g500-logn20	3776.22	106.36	15.12	**8.35**	17.37	10.19
kron_g500-logn21	9514.79	197.58	32.69	**18.98**	31.16	21.64
soc-LiveJournal1	252	113.88	19.85	**22.75**	25.74	25.83
uk-2002	4229.68	853.09	96.68	**58.31**	146.47	103.75
uk-2005	1971.65	2144.96	223.86	**136.44**	293.45	204.81

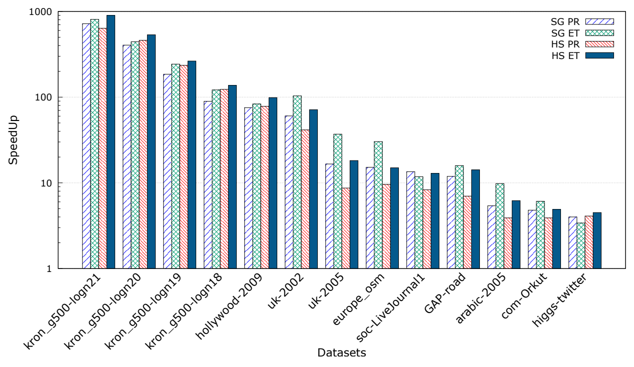


Figure 3: Speedup: Our algorithms w.r.t static GPU BFS for insert

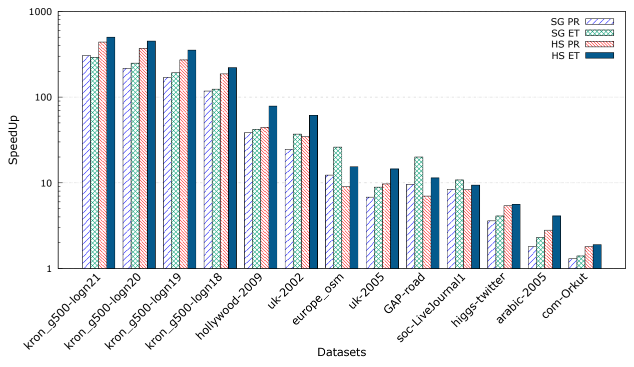


Figure 4: Speedup: Our algorithms w.r.t static GPU BFS for delete

6.4 Scalability Analysis: Impact of Increasing Batch Sizes on Performance.

After evaluating the speedups of our proposed algorithms, we now focus on studying their scalability. Specifically, we aim to understand the performance of our algorithms by varying the batch size.

We conducted experiments with batch sizes of 100, 1,000, 10,000, and 100,000 edges and running times are reported in Figures 5 and 6. The results demonstrate that our algorithms maintain strong scalability, with minimal variation in running times across different batch sizes, achieving an average throughput of 2M insertions and

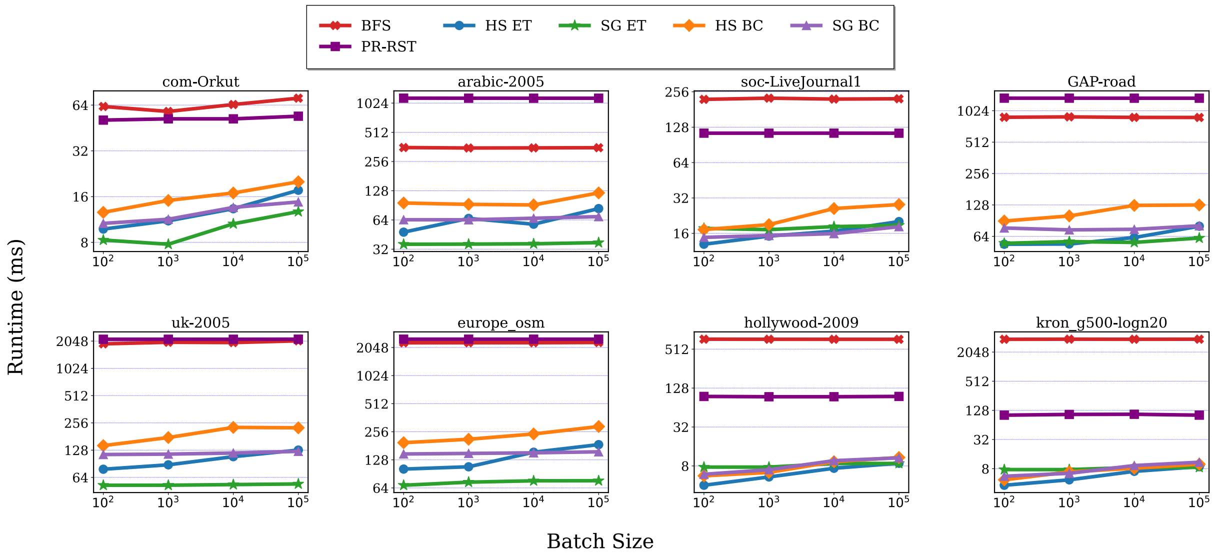


Figure 5: Insertion operation: Performance comparison of baseline and proposed algorithms across different batch sizes.

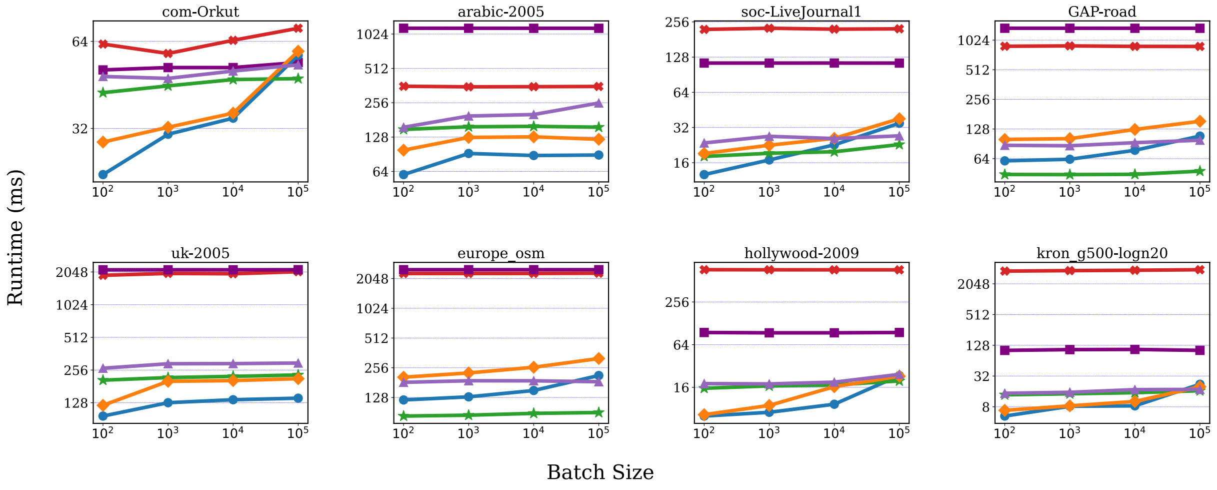


Figure 6: Deletion operation: Performance comparison of baseline and proposed algorithms across different batch sizes.

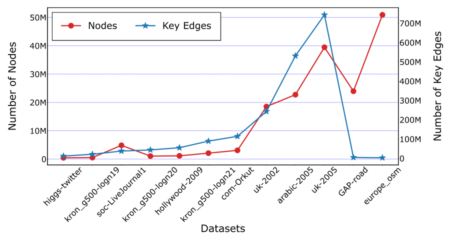
1.4M deletions per second across all batch sizes respectively. This suggests that our algorithms can efficiently handle larger batches without significant performance degradation.

6.5 Influencing Factors in proposed algorithms

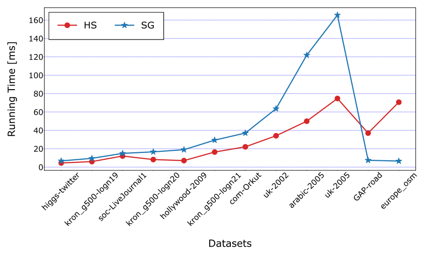
In this section, we analyze the factors influencing the performance of our algorithms. While all algorithms demonstrated significant speedups and scalability, running times varied across different batches and operations. We now identify the key factors that contribute to these variations.

Although deletions theoretically introduce an $O(\log E)$ overhead, our experiments reveal that this accounts for only 10-15% of the total runtime. This is largely because the compaction algorithm is highly optimized for GPUs due to its effective parallelization techniques [9]. However, most of the time (approximately 50%) was spent on identifying oriented replacement edges, while path reversal accounted for around 30%, except for insertions in SG-based algorithms. Interestingly, despite both broadcasting and the Eulerian Tour having the same $O(\log n)$ complexity for *Path Reversal*, the latter demonstrated superior practical performance. We now explore the rationale behind these observed behaviors.

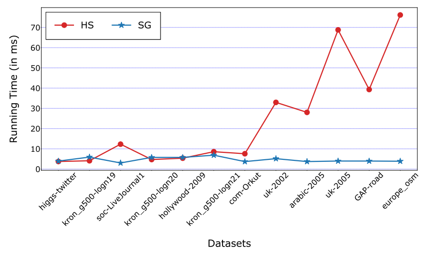
Hooking-Shortcutting vs SuperGraph. Figure 7 summarizes the performance trends of Hooking-Shortcutting (HS) and SuperGraph (SG) across deletion and insertion operations, highlighting how their running times are influenced by the number of vertices and key edges, respectively. For the delete operation, as illustrated in Figures fig. 7a and 7b, it is evident that the performance of HS depends on the number of vertices (n), whereas the performance of SG is influenced by the number of key edges (m). Specifically, HS performs better than SG when $n < m$, otherwise vice versa.



(a) Vertices vs. key edges for deletion (batch size: 10,000).



(b) Running times for deletion operations.



(c) Running times for insertion operations.

Figure 7: HS performance depends on the number of nodes (n), whereas SG performance depends on the number of key edges (m). For insertion, the number of key edges equals the batch size, which is fixed across datasets.

Next, turning to the insert operation, Figure fig. 7c shows that the running time of SG remains nearly constant, since the number of key edges equals the batch size during insertion, and the batch size is fixed across our experiments. In contrast, the running time of HS varies with the number of vertices. Consequently, SG consistently outperforms HS for insertion operations.

Thus, we conclude that HS is more suitable when the number of vertices is significantly smaller than the number of key edges, while SG is preferable when the number of key edges is substantially smaller than the number of vertices.

We next examine the performance differences between Broadcasting and Eulerian Tour for path reversal.

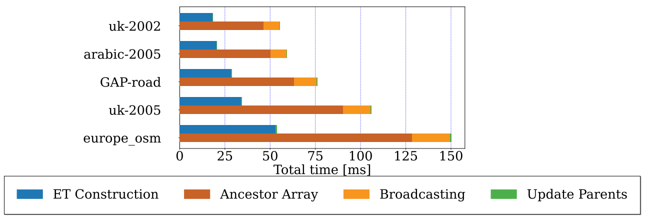


Figure 8: Running time breakdown of path reversal algorithms

Broadcasting vs Eulerian Tour. The Eulerian Tour technique outperforms Broadcasting by a factor of $2-3\times$, as shown in Figure 8. This performance gap is primarily due to the overhead caused by the excessive number of uncoalesced memory operations during the creation of the ancestor array in the broadcasting method. The pointer jumping steps, which identify all the ancestors, involve each vertex v writing to the $(v \times \log n + i)^{th}$ location in the *special ancestor*$[\cdot]$, where i is the iteration. This process occurs $\log n$ times, resulting in a significant number of uncoalesced memory operations.

Additionally, the creation of the *on-path* array introduces further overhead of $\log n$ and uncoalesced memory reads, as all active vertices traverse its special ancestor array repeatedly during $\log n$ iterations. Although the update of the parent occurs in constant time, these factors significantly impact overall performance.

7 Discussion

For dense graphs, the supergraph-based approach for deletions can become inefficient because the number m of key edges can reach $O(n^2)$. In contrast, the Hooking-Shortcutting approach, being independent of m, is less affected by graph density and maintains stable performance. For insertions, supergraph-based approaches perform efficiently even in dense graphs, as their performance relies solely on processing key edges—the batch of edges to be inserted constitutes the key edges.

8 Conclusion

We designed and implemented fully dynamic updates of rooted spanning trees in a many-core (GPU) environment. Our algorithms focus on repairing the spanning tree rather than reconstructing it from scratch, using key techniques such as hooking and shortcutting, supergraph-based methods for identifying oriented replacement edges, and Eulerian tour-based path reversal. These optimizations resulted in significant performance improvements over existing approaches.

References

[1] Umut A Acar, Daniel Anderson, Guy E Blelloch, and Laxman Dhulipala. 2019. Parallel batch-dynamic graph connectivity. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 381–392.

[2] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. 587–596.

[3] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[4] Guojin Cong and David A Bader. 2004. The Euler tour technique and parallel rooted spanning tree. In *International Conference on Parallel Processing, 2004. ICPP 2004*. IEEE, 448–457.

[5] Quinten De Man, Laxman Dhulipala, Adam Karczmarz, Jakub Łącki, Julian Shun, and Zhongqi Wang. 2024. Towards Scalable and Practical Batch-Dynamic Connectivity. *arXiv preprint arXiv:2411.11781* (2024).

[6] 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.

[7] 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.

[8] Pawan Harish and Petter J Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*. Springer, 197–208.

[9] Mark Harris. 2007. *Parallel Prefix Sum (Scan) with CUDA*. Addison-Wesley, Chapter 39, 851–876. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

[10] Chirayu Anant Haryan, G Ramakrishna, Kishore Kothapalli, and Dip Sankar Banerjee. 2022. Shared-memory parallel algorithms for fully dynamic maintenance of 2-connected components. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1195–1205.

[11] 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.

[12] Joseph JáJá. 1992. *Parallel algorithms.* Addison Wesley Longman Publishing Co., Inc.

[13] Arindam Khanda, Sriram Srinivasan, Sanjukta Bhowmick, Boyana Norris, and Sajal K Das. 2021. A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 929–940.

[14] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.

[15] Robert McColl, Oded Green, and David A Bader. 2013. A new parallel algorithm for connected components in dynamic graphs. In *20th Annual International Conference on High Performance Computing*. IEEE, 246–255.

[16] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.

[17] Nosferalatu. 2020. Simple GPU Hash Table. In `nosferalatu.com`. https://nosferalatu.com/SimpleGPUHashTable.html

[18] Adam Polak, Adrian Siwiec, and Michał Stobierski. 2021. Euler meets GPU: practical graph algorithms with theoretical guarantees. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 233–244.

[19] Sanguthevar Rajasekaran and John H Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.* 18, 3 (1989), 594–607.

[20] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB journal* 29 (2020), 595–618.

[21] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[22] 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.

[23] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 1–8.

[24] Sriram Srinivasan, Sanjukta Bhowmick, and Sajal Das. 2016. Application of graph sparsification in developing parallel algorithms for updating connected components. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 885–891.

[25] Sriram Srinivasan, Samuel D Pollard, Boyana Norris, Sajal K Das, and Sanjukta Bhowmick. 2018. A shared-memory algorithm for updating tree-based properties of large dynamic networks. *IEEE Transactions on Big Data* 8, 2 (2018), 302–317.

[26] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. 2019. Batch-parallel euler tour trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 92–106.

[27] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), 1–49.