Towards Scalable and Practical Batch-Dynamic Connectivity

Quinten De Man
University of Maryland
deman@umd.edu

Laxman Dhulipala
University of Maryland
laxman@umd.edu

Adam Karczmarz
University of Warsaw & IDEAS NCBR
a.karczmarz@mimuw.edu.pl

Jakub Łącki
Google Research
jlacki@google.com

Julian Shun
MIT CSAIL
jshun@mit.edu

Zhongqi Wang
University of Maryland
zqwang@umd.edu

ABSTRACT

We study the problem of dynamically maintaining the connected components of an undirected graph subject to edge insertions and deletions. We give the first parallel algorithm for the problem that is work-efficient, supports batches of updates, runs in polylogarithmic depth, and uses only linear total space. The existing algorithms for the problem either use super-linear space, do not come with strong theoretical bounds, or are not parallel.

On the empirical side, we provide the first implementation of the *cluster forest algorithm*, the first linear-space and polylogarithmic update time algorithm for dynamic connectivity. Experimentally, we find that our algorithm uses up to 19.7× less space and is up to 6.2× faster than the level-set algorithm of Holm, de Lichtenberg, and Thorup, arguably the most widely-implemented dynamic connectivity algorithm with strong theoretical guarantees.

PVLDB Reference Format:
Quinten De Man, Laxman Dhulipala, Adam Karczmarz, Jakub Łącki, Julian Shun, and Zhongqi Wang. Towards Scalable and Practical Batch-Dynamic Connectivity. PVLDB, 18(3): 889 - 901, 2024.
doi:10.14778/3712221.3712250

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ParAlg/DynamicConnectivity.

1 INTRODUCTION

The problem of dynamically maintaining the connected components of a graph is a critical subroutine that is often used when designing dynamic algorithms for other fundamental and practical problems, e.g., dynamic DBSCAN [21, 31], hierarchical clustering [38, 43], approximate MST [17], among other problems. It is also one of the most intensely studied dynamic graph problems, and has seen extensive algorithmic development over the past three decades [18, 24–27, 29, 30, 39, 44, 45, 47, 48]. In the fully-dynamic graph connectivity problem, the goal is to build a data structure that supports the following operations on an undirected graph G:

- **Insert**(u, v) inserts edge (u, v) into G.
- **Delete**(u, v) deletes edge (u, v) from G.
- **Connected**(u, v) returns whether u and v are connected in G.

Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712250

Despite its importance, dynamic connectivity has not yet been efficiently solved in practice, and marks a major gap in our understanding of how to bridge theory and practice in dynamic algorithms. In the special case of dynamic forest-connectivity, there are data structures, such as *Euler Tour Trees (ETTs)* [24], that are reasonably practical and have been adapted to support parallel updates [46]. However, few implementations exist for the more complex case of general graphs. For general graphs, perhaps the best known dynamic connectivity data structure that provides good theoretical guarantees and has been implemented [28] is due to Holm, de Lichtenberg, and Thorup [26], who developed a dynamic connectivity algorithm that performs updates in $O(\log^2 n)$ amortized time and requires $O(n \log n + m)$ space, for a graph with n vertices and m edges. Their algorithm is based on $O(\log n)$ layers of dynamic forest-connectivity data structures; we refer to their idea as the *HLT algorithm*.

However, existing implementations of the HLT algorithm suffer from high overheads in space and time, limiting their practical applicability. For example, we found that to run on a 1.2 billion edge graph, an optimized dynamic connectivity implementation based on the HLT algorithm requires up to 360 billion bytes—over 70 times more than what it takes to store the graph using a simple (static) representation. The key limitation of implementations based on the HLT algorithm is that the connectivity information is stored redundantly in separate forest-connectivity data structures across a logarithmic number of different layers. At a high level, the HLT algorithm maintains a spanning forest F and a hierarchy of nested edge subsets $F_1 \subseteq F_2 \subseteq \ldots F_k = F$. With this representation, the vertices of G are present in k trees, which results in a space usage of $O(n \log n)$ across all of the trees.

On the theoretical side, the space usage was improved to linear by an elegant *cluster forest algorithm (CF algorithm)* that is inspired by the HLT algorithm [45]. The key idea in the CF algorithm is to store a single forest of trees (called the *cluster forest*) that implicitly represents the connectivity information stored in the nested $O(\log n)$ layers of the HLT algorithm. In its basic version, the CF algorithm achieves similar update times to HLT, while improving the space bound to $O(n + m)$. This was the first algorithm that solved dynamic graph connectivity in polylogarithmic update time and linear space. The CF algorithm was later simplified and optimized by Wulff-Nilsen [47].

Given the theoretical advantages of the CF algorithm, an important question is: ***is the CF algorithm practical?*** For example, does it yield improved space-efficiency or faster runtime in practice relative to existing dynamic connectivity implementations? This question is highly non-trivial due to the complexity of implementing

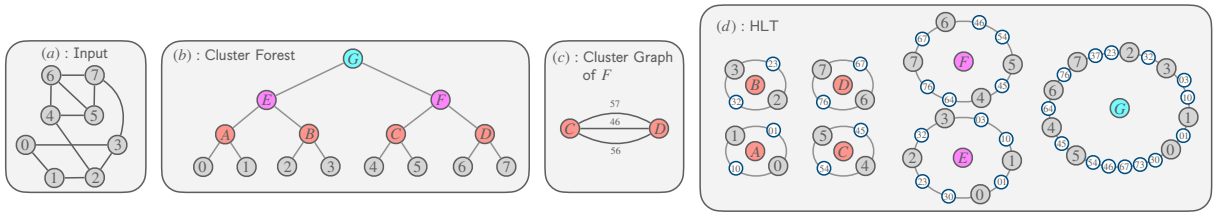

Figure 1: The core data structures used by the cluster forest (CF) and HLT algorithms. The input graph is shown in (a). The cluster forest is given in (b), and represents the nested hierarchy of connected components. (c) shows the cluster graph of the level 2 component F. Lastly, (d) shows the same component hierarchy as (b), but as stored by the HLT algorithm.

and optimizing the CF algorithm, which uses a significant amount of indirection and requires performing sophisticated tree traversals and amortization to obtain its update bounds. A second important question is: ***can the CF algorithm be efficiently parallelized?*** In particular, can we ensure that each update is processed with low *depth* (longest chain of sequentially dependent instructions) in the worst case? Furthermore, can we also make the algorithm work-efficient in the *parallel batch-dynamic setting*? We note that the batch-dynamic setting, in which updates come in batches of arbitrary size, is the standard modern setting for parallelizing dynamic algorithms [1, 5, 16, 20, 23, 35, 41]. Ideally, we would like to parallelize the algorithm without sacrificing space-efficiency or work-efficiency. That is, each batch of updates should be performed with low depth, and work (total number of operations) and space matching that of the sequential CF algorithm. We note that while the HLT algorithm was recently shown to be amenable to an efficient batch-dynamic algorithm [1], it is not space-efficient.

In this paper, we carefully study the CF approach in theory and practice to answer these open questions. On the theoretical side, we extend the CF algorithm to the parallel setting and show how to achieve low depth. Specifically, we introduce a new invariant (the *blocked invariant*), which provides important additional structure that we exploit. Using our new invariant and our approach to maintaining it, we obtain the first space-efficient and work-efficient parallel algorithm that has polylogarithmic depth.

On the empirical side, we perform the first experimental study of the CF approach in the sequential setting. Compared with the existing state-of-the-art dynamic connectivity implementations with worst-case guarantees based on the HLT algorithm, we find that our implementations use up to 19.7× less space and are up to 6.2× faster than an optimized implementation of HLT. In the next two sections, we formalize the data structures, and present a technical overview of our results.

2 PRELIMINARIES

Model. We use the work-depth (or work-span) model for fork-join parallelism to analyze parallel algorithms [7, 13]. The model assumes a set of threads that share memory. A thread can `fork` k child threads that run in parallel. When all children complete, the parent thread continues. The ***work*** W of an algorithm is the total number of instructions and the ***depth*** (span) D is the length of the longest sequence of dependent instructions. Computations can be executed using a randomized work-stealing scheduler in practice in $W/P + O(D)$ time with high probability on P processors [2, 8, 22].

Definitions. We start by introducing definitions used throughout the paper when describing the dynamic connectivity algorithm. We are given an undirected graph G, with vertices V and edges E. We use n to denote the number of vertices and m to denote the number of edges. Each edge $e = (u, v) \in E$ has a level $l(e) \in [1, L_{\max}]$ assigned to it, where $L_{\max} = \lceil \log n \rceil$. Let $E_i = \{e = (u, v) \in E \mid l(e) \leq i\}$ be the set of all edges with levels $\leq i$. Let $G_i = (V, E_i)$. We maintain the following size invariant on the connected components of each G_i:

Invariant 2.1 (Size Invariant). *The maximum size of a connected component of G_i is 2^i.*

We can imagine contracting each of the connected components of G_i to obtain V_i, the set of ***clusters*** at level i. The ***children*** of a cluster $c \in V_i$ are the clusters in V_{i-1} that are merged together using level i edges to obtain c.

Cluster Forest. By explicitly representing the relationship between clusters on different levels, we obtain the ***cluster forest***, which we denote using C, in which each node has a level in $[0, L_{\max}]$, and where the nodes at level i represent the clusters of connected components of G_i (the graph containing all edges with level $\leq i$). The root(s) of C are nodes representing clusters at level $L_{\max}$ in G and correspond to the connected components of the graph. The leaves of C are nodes representing the clusters at level 0 in G, and correspond to the original vertices of the graph. Figure 1(b) illustrates the cluster forest for the graph in Figure 1(a). If clusters with only a single child are not stored, it is not difficult to see that the number of nodes in C is $O(n)$.

Each node v in C also stores the size of the cluster that it represents, $n(v)$, which is equal to the number of leaves in the subtree rooted at v ($n(v) = 1$ for leaf nodes). By the size invariant above (Invariant 2.1), for any level i cluster c, we have $n(c) \leq 2^i$.

For a cluster $c \in C$ at level i, the ***cluster graph***, $CG(c)$ of the node is the graph formed by taking its child clusters at level $(i-1)$ as the vertices, and where the edges are the level i edges incident to all leaf vertices in c. Figure 1(c) illustrates the cluster graph for a vertex in the cluster forest illustrated in Figure 1(b).

We define a ***self-loop edge*** as a level i edge for which its endpoints are contained within the same level $(i-1)$ cluster. Note that a level i self-loop edge on a level $(i-1)$ cluster C differs from a level $(i-1)$ edge with both endpoints in C. The level i self-loop edge appears in the cluster graph of the parent of C as a self-loop on C (thus the name). The level $(i-1)$ edge appears in the cluster graph of C and connects two children of C.

3 TECHNICAL OVERVIEW

Parallelizing the CF Approach with Low Depth. In both the CF and HLT approaches, *replacement edge search*, i.e., the search for an edge that certifies the connectivity between the endpoints of a deleted edge (i.e., "replaces" it) is the most complex aspect of the data structure. Both algorithms maintain a hierarchy of nested edge subsets $E_1 \subseteq E_2 \subseteq \ldots \subseteq E_k$, with the HLT approach storing a spanning tree of each subset, and the CF algorithm using a more space-efficient representation (see Figure 1). In both algorithms, replacement search is handled by carefully searching the nested hierarchy of components from the lowest level component containing the deleted edge to the highest level. For example, in Figure 1, if the edge 4–6 is deleted, the edges in component F will be searched, and either 5–6 or 5–7 can be used as a replacement edge. Both algorithms maintain the *size invariant* (Invariant 2.1), which ensures that components at level i have size at most 2^i. In each component, the non-tree edges are searched to find a replacement edge, and the unsuccessfully searched edges are pushed to a lower level to pay for the cost of searching them. The parallel version of the HLT algorithm by Acar et al. [1] obtains parallelism by performing a doubling search over the non-tree edges incident to the smaller of the two components induced by the deleted edge (obtaining the smaller component, and indexing the non-tree edges is made possible using Euler Tour Trees). Note that it is critical that the edges searched are incident to the *smaller* component, since this component (and its non-tree edges) will be pushed to a lower level to pay for the search.

Unlike the HLT algorithm, which maintains an Euler Tour Tree for every component (see Figure 1(d)), and can easily split a component into the smaller/larger halves by deleting the edge in the ETT, the CF algorithm only has access to the *cluster graph* of the component that an edge is deleted from, due to being more space-efficient. Recall that the cluster graph of a level i node consists of *all edges* at level i (all such edges go between the level $i-1$ children of this node). Since the CF algorithm cannot simply remove the edge and split the component into smaller/larger halves, the algorithm performs a more careful *graph search* that effectively interleaves two searches (e.g., breadth-first searches) from both level $i-1$ clusters incident to the deleted edge. Like in the HLT algorithm, level i edges that are unsuccessfully inspected can be paid for by pushing them down to level $i-1$. We give more details in Section 4.

The graph search, which needs to discover the connected component of a graph undergoing changes, seems extremely difficult to parallelize work-efficiently using the existing CF algorithm. The main issue is that the graph in the cluster forest at this level can have a very high diameter; in fact, the diameter can be as high as $\Theta(n)$, making a work-efficient parallelization of this process very challenging unless we are willing to sacrifice having low depth.

We solve this problem by introducing a new invariant called the *blocked invariant* that ensures that the cluster graph stored at every internal node in the cluster forest is *guaranteed to have constant diameter* (in fact the diameter is always at most 2). The key idea of the invariant is simple to state: we ensure that every cluster is incident to at least one edge that cannot be further pushed down to a lower level without violating the size invariant. We prove that this property implies that the cluster graph at each node is guaranteed to have diameter at most 2. As a result, during replacement search any parallel graph procedure (e.g., parallel breadth-first search) will suffice, and help us obtain low depth, since the diameter is low. However, maintaining this property dynamically turns out to be very tricky, even in the sequential setting; our main algorithmic contributions are novel sequential and parallel algorithms to maintain the invariant under (batch-)dynamic updates. Overall, our new parallel batch-dynamic algorithm can perform insertions and deletions with $O(\log^2 n)$ amortized work per edge update and $O(\log^3 n)$ depth per batch.

The CF Approach in Practice. In addition to our theoretical contributions, in this paper we give the first implementations of any cluster forest data structure, and study the practicality of the CF approach and our new invariants in the sequential setting. As we discuss in more detail in Section 8, implementing CF algorithms seems to be even more involved than implementations of the HLT algorithm. A significant implementation challenge is that a single data structure—the cluster forest—stores both the hierarchy of connected components and the non-tree edges stored at each component in the hierarchy (in the HLT algorithm, the implementation complexity is somewhat lower since each component is stored as a separate Euler Tour Tree). Our experiments show that our new implementations of the CF approach are significantly more space-efficient and process updates faster than existing state-of-the-art dynamic connectivity implementations. For example, across a diverse set of graph inputs, our CF implementations achieve up to 19.7× lower space usage and up to 6.2× faster updates compared to a carefully designed implementation of HLT.

4 SEQUENTIAL CF ALGORITHM

Next, we give a more detailed overview of how the sequential CF algorithm performs insertions and deletions.

Insert $e = (u, v)$. The CF algorithm first sets the level of the edge, $l(e) = L_{\max}$. Let r_u and r_v be the roots of the trees containing u and v, respectively, in C. If $r_u = r_v$, then nothing further needs to be done. If $r_u \neq r_v$, then we have increased the connectivity of the graph and must merge r_u and r_v together. This requires a *merge* operation on C that takes the roots of two trees and merges them together by adding the children of (without loss of generality) r_v as children of r_u and deleting r_v.

Delete $e = (u, v)$. As in the HLT algorithm, deletion requires performing a *replacement edge search* to check if the deletion of (u, v) affects the connectivity of G. Let $i = l(e)$. In the CF algorithm, the problem boils down to *certifying the connectivity* of the level i cluster, P, containing u and v. Recall that the cluster graph of P consists of the level $(i-1)$ clusters that are children of P and the level i edges with both endpoints in P. Let the level $(i-1)$ clusters containing u and v be C_u and C_v, respectively, and the cluster graph of P, $CG(P)$, be G_i. If $C_u = C_v$ (e is a self-loop), then we can quit since the connectivity of G_i is unaffected after deleting e.

If $C_u \neq C_v$, then we need to check whether (u, v) was a *bridge* of G_i (i.e., every possible spanning tree of G_i must use this edge). If the deleted edge is a bridge, then G_i splits into two components. This means that the cluster forest C must be updated, and then the algorithm must recursively check at level $(i+1)$ whether the two split pieces of G_i can be reconnected using a level $(i+1)$ edge.

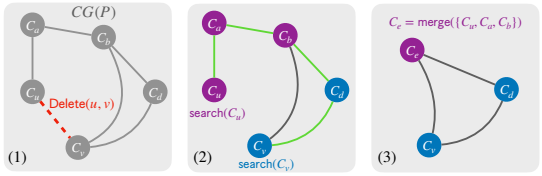

Figure 2: (1): The cluster graph of the level i cluster P containing a deleted level i edge (u, v); the vertices of the cluster graph are the level $(i-1)$ child clusters of P. (2): Deleting (u, v) may disconnect $CG(P)$, so search(C_u) and search(C_v) are run to check if C_u and C_v are still connected using level i edges in $CG(P)$. Green edges are edges explored during the search. (3): In this example, $CG(P)$ remains connected, $\{C_u, C_a, C_b\}$ are merged, and the explored level i edges of the smaller search are pushed down to level $(i-1)$.

To certify connectivity in G_i, the algorithm runs two searches from C_u and C_v using any graph search procedure, such as breadth-first search. Call these searches search(C_u) and search(C_v). The searches explore the multi-graph of level $(i-1)$ clusters that are children of P, and level i edges between them (see Figure 2). Unlike in the HLT algorithm, the CF algorithm does not know which of C_u or C_v is in the smaller component after deleting (u, v). To obtain good amortized work bounds, the CF algorithm alternates between steps of search(C_u) and search(C_v). The searches stop when either (1) a common vertex in G_i is explored by both searches, or (2) one of the searches runs out of level i edges to explore, certifying that C_u and C_v are no longer connected using level i edges.

In case (1), C_u and C_v must be connected so G_i is unaffected, and thus the cluster forest $\mathcal{C}$ does not change. Let S_u and S_v be the set of level $(i-1)$ clusters explored by u and v's searches, respectively. Let $n_u = \sum_{c \in S_u} n(c)$ and $n_v = \sum_{c \in S_v} n(c)$. Then $\min(n_u, n_v) \leq 2^{i-1}$, and so we can push all of the level i edges explored by the smaller search down to level $(i-1)$. The alternating search ensures that we do not need to worry about pushing down edges incident to the larger component as the work on these edges is paid for by the pushing down of edges in the smaller component. Pushing edges to level $(i-1)$ can require merging level $(i-1)$ clusters.

In case (2), we take the search with a smaller total size value, without loss of generality search(C_u), and push all of the level i edges it explored down to level $(i-1)$ to pay for the search. This requires merging all level $(i-1)$ clusters explored by search(C_u). Let W be this new level $(i-1)$ node that they were all merged into. Next, we need to update $\mathcal{C}$ to reflect the fact that G_i split. We first remove W as a child of P in $\mathcal{C}$ and decrement $n(P)$ by $n(W)$. We then create a new level i node P' and set the parent of W to P', and add P' as a child of the parent of P. Only in case (2) do we need to continue (recursively) at level $(i+1)$ to check if the level i clusters containing u and v remain connected using the level $(i+1)$ edges, or whether a similar split needs to happen at level $(i+1)$ or above.

Cluster Forest Interface. Studying this algorithm, we can identify the following necessary operations on the cluster forest $\mathcal{C}$:

(1) **FetchEdge**(C, i) the search needs to be able to iterate over the level i edges incident to a level $(i-1)$ cluster C.
(2) **Parent**(C) returns the parent of C in the cluster forest.
(3) **AddChild**(P, C) adds a node C as a child of node P.
(4) **RemoveChild**(P, C) removes C from the children of P.
(5) **Cluster**(v, l) returns the level l cluster containing v.
(6) **Merge**(C_1, C_2) merges two level $(i-1)$ clusters C_1 and C_2.
(7) **PushDown**(e) pushes down a level i edge e.

Local Trees Implementation. Since a node in the cluster forest may have at most n children, the CF algorithm represents the cluster graph of each cluster in $\mathcal{C}$ using *local trees* to allow performing all of the above operations in $O(\log n)$ time. A local tree for a cluster u in $\mathcal{C}$ is a binary tree whose leaves are the children of u in $\mathcal{C}$. Let the rank of a child v of u be $r(v) = \lfloor \log n(v) \rfloor$. Initially, each child cluster is in its own tree. While there are two trees r and r' of the same rank, they are paired up into a new tree r'' with rank one larger. Once this pairing process terminates, there are at most $\log n$ trees $T_1, \ldots, T_k$, which are called ***rank trees***. The collection of rank trees are combined into a single binary tree by connecting them in order of rank along the right spine of a binary tree.

To efficiently iterate over level i edges during the search, the trees are augmented using a $\log n$-length *bitmap* (stored as a single word), where the i-th bit is 1 if and only if there is a level i edge incident to some leaf vertex in the subtree. All of the operations needed in the sequential CF algorithm can be implemented in $O(\log n)$ worst-case time using local trees [47].

Advantages and Challenges of the CF Algorithm. One immediate advantage of the cluster forest algorithm [47] is that the space requirement for the cluster forest can be made $O(n+m)$ by simply ensuring that the cluster forest is path compressed, i.e., a level-i cluster c is explicitly represented if and only if either $i = 0$ or there is a level i edge e with both endpoints in c (thus either e is a self-loop or c has at least two level-$(i-1)$ child clusters). On the other hand, the HLT algorithm requires $O(n \log n + m)$ space, since each vertex is potentially present in an Euler Tour Tree at all $O(\log n)$ levels. Since large graphs in practice are often extremely sparse [15], achieving linear total space is an important goal that can lead the community towards practical and theoretically efficient implementations of dynamic graph connectivity.

As discussed in Section 3, the main challenge with parallelizing the CF algorithm is how to perform the replacement edge search, which requires work-efficiently traversing the cluster graph (which can potentially have very high diameter).

5 THE BLOCKED CLUSTER FOREST

We will use the idea of *blocked edges* to obtain more structured cluster graphs that have bounded diameter and enable us to parallelize the connectivity search. Here we define blocked edges and establish the main invariant of our new data structure:

Definition 5.1 (Blocked Edge). A level i edge $e = (u, v)$ is a ***blocked edge*** if it cannot be pushed to level $(i-1)$ without violating Invariant 2.1. That is, a level i edge (u, v) is blocked if and only if $n(\text{cluster}(u, i)) + n(\text{cluster}(v, i)) > 2^{i-1}$. An ***unblocked edge*** is an edge that is not blocked.

Invariant 5.2 (Blocked Edge Invariant)**.** *Consider any cluster graph $CG(c)$ of a level i cluster c. Then every level $(i-1)$ cluster $X \in CG(c)$ is incident to at least one blocked level i edge or c is an isolated cluster and only has one child X.*

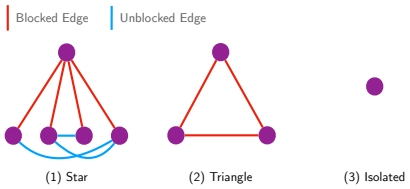

Figure 3: The three possible cases for what the cluster graph of a level i cluster in a blocked cluster-forest can look like.

A ***blocked cluster-forest*** is defined as a cluster forest where every cluster satisfies Invariant 5.2. An ***isolated cluster*** in the cluster forest is a cluster that has only a single child. The blocked edge invariant is useful since we can show that it implies that every cluster graph has low diameter, making them more amenable to parallel search. Next, we describe several important and useful properties of the blocked cluster-forest. The proofs are left to the full paper [36]. The key property is that a maximum matching computed over the blocked edges must have size at most 1, as summarized by the following lemma:

Lemma 5.3. *Suppose Invariant 5.2 holds for a cluster graph $CG(c)$ of a level i cluster c. Let M be the size of the maximum matching in $CG(c)$ over only the blocked edges. Then $M \leq 1$.*

The invariant also implies that we cannot have a path using only blocked edges of length ≥ 3, since such a path has a blocked matching of size $M > 1$. This allows to prove the following lemma that bounds the diameter of any cluster graph.

Lemma 5.4. *Suppose Invariant 5.2 holds for a cluster graph $CG(c)$ of a level i cluster c. Then $CG(c)$ has diameter $\mathsf{diam}(CG(c)) \leq 2$.*

The following two lemmas describe other properties of the blocked cluster forest which enable efficient updates:

Lemma 5.5. *Suppose Invariant 5.2 holds for a graph $CG(c)$ of a level i cluster c. Then, there exists a **center** node in $CG(c)$ that is connected to every other node by a blocked edge.*

Lemma 5.6. *Suppose Invariant 5.2 holds for a graph $CG(c)$ of a level i cluster c, and $CG(c)$ has $k \geq 4$ nodes. Then, the center node corresponds to the largest cluster in $CG(c)$.*

Using Structure for Parallel Connectivity Search. Consider a level i cluster c. We can characterize $CG(c)$ as one of following:

- The cluster graph is a star (case (1) in Figure 3).
- The cluster graph is a triangle (case (2) in Figure 3).
- The cluster graph is a single node (case (3) in Figure 3).

Figure 4 illustrates the relationship between clusters in the cluster forest C and the cluster graph of a node. Since the graphs are guaranteed to have low diameter, performing a graph search on $CG(c)$ from two clusters C_u and C_v can be done in low depth by running the searches in lock step and doubling the number of edges that we explore at each step. The doubling ensures that we can still amortize the exploration cost to level decreases on the edges in previous steps while ensuring that the search runs in polylogarithmic depth. The main challenge now is how to maintain the blocked invariant dynamically.

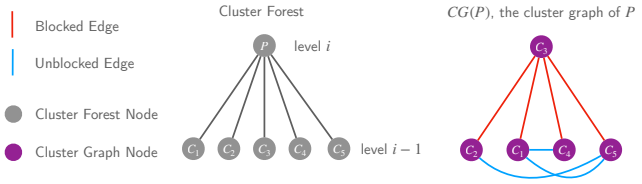

Figure 4: Illustration of the cluster graph for a node in the blocked cluster-forest, and its star structure. The center node of $CG(P)$ is C_3, and all other nodes are satellites connected to the center through a blocked edge.

5.1 Updating the Blocked Cluster-Forest

In this section, we describe how to maintain Invariant 5.2 while performing single edge insertions and deletions. Surprisingly, achieving this goal even in the sequential setting requires some non-trivial ideas and analysis.

Pushing Edges Down. Note that pushing an edge from level $(i+1)$ to i may violate the blocked invariant for its new level $(i-1)$ endpoints. Lemma 5.7 and 5.8 prove that if an unblocked edge is *repeatedly* pushed down until it is blocked, then the blocked invariant will be maintained. The proofs are left to the full paper [36] due to space constraints.

Lemma 5.7. *If a level i edge e between two distinct level $(i-1)$ clusters C_1 and C_2 is unblocked, then either C_1 or C_2 is an isolated cluster, i.e., its cluster graph consists of a single level $(i-2)$ cluster.*

Lemma 5.8. *If an unblocked edge is pushed down until it is blocked then Invariant 5.2 is preserved.*

Insert $e = (u, v)$. We give a simple top-down algorithm for insertion. In the cluster forest we include a single global level $L_{\max}+1$ cluster whose children are the roots of all the components. Then for insertion, add the edge as a level $(L_{\max}+1)$ edge. No level $(L_{\max}+1)$ edge can be blocked as the size constraint for level $L_{\max}$ clusters is $\geq n$. Next repeatedly push down e until it is blocked. Lemma 5.8 proves that this maintains the blocked invariant.

Delete $e = (u, v)$. Let the level of e be $i = l(e)$ prior to deletion. Similar to before, let $C_u = \mathsf{cluster}(u, i-1)$ and $C_v = \mathsf{cluster}(u, i-1)$, the two level $(i-1)$ clusters containing u and v. Let P be the parent cluster of C_u and C_v, and GP be the grandparent. Let $G_i = CG(P)$ be the cluster graph of P. Like before, if $C_u = C_v$ then the edge is a self-loop we can quit here since the connectivity is unaffected. If the edge is unblocked it can be safely deleted because either it is a self loop or the clusters are connected by 1 or 2 blocked edges. If $C_u \neq C_v$ and the edge is blocked, deletion of the edge may cause C_u and C_v to be in violation of the blocked invariant.

Consider a level i edge. Any unblocked level i edges that we encounter can be pushed down (repeatedly until blocked). Any blocked level i edges must remain at this level. Since there can be multiple (parallel) edges between two level $(i-1)$ clusters that are all blocked, we need to be careful that we don't process too many blocked edges, since we can't push these down. To restore the blocked invariant at this level (the lowest level that the deletion affects), we repeatedly fetch and push down a level i edge incident to C_u or C_v in an alternating fashion until we have certified that

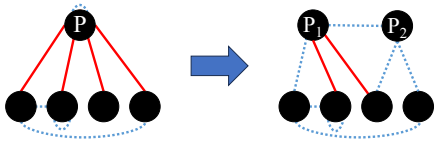

Figure 5: A cluster graph before and after the center cluster P splits. The edges incident to P are divided between P_1 and P_2.

both are incident to a blocked edge (or have no more incident edges). Restoring the invariant at higher levels turns out to be more involved, and we describe this shortly.

Once we have restored the invariant at this level, we can certify the connectivity of C_u and C_v as follows. If both have no more incident edges, then they are connected if and only if $C_u = C_v$. If one has no more incident edges and the other is incident to a blocked edge, they must not be connected. If both are incident to a blocked edge, they must be connected (otherwise the two blocked edges form a matching of size 2 over the blocked edges in $CG(P)$).

Just as in original CF algorithm, if the connectivity search failed at level i, the algorithm must continue to level $(i+1)$ and perform connectivity search over the level $(i+1)$ edges. In this case, one of the components must have merged into a single level $(i-1)$ cluster. Let W be this cluster. We (1) remove W as a child of P and decrement $n(P)$ by $n(W)$, (2) create a new level i cluster P', set the parent of W to P', and (3) add P' as a child of GP.

When W is removed as a child of P and added as a child of a new cluster P', P is essentially being split into two clusters, P_1 and P_2 (see Figure 5). The size of P is split between P_1 and P_2. Any edges previously incident to P are now incident to either P_1 or P_2 (or possibly both if it was a self-loop). Since the sizes of both P_1 and P_2 are less than that of P, these edges may now be unblocked, thus the cluster graph of GP potentially violates the blocked invariant.

Restoring the Blocked Invariant at Higher Levels. Unlike restoring the blocked invariant at the lowest level that an edge was deleted, which is straightforward, restoring the invariant at higher levels is much trickier since clusters can be split as illustrated in Figure 5. If the number of clusters in $CG(c)$ is bounded by some constant, we can afford to restore the invariant by checking that every single cluster is incident to a blocked edge, pushing any found unblocked edges down.

Otherwise (if the number of clusters is $\omega(1)$), Lemmas 5.5 and 5.6 tell us that there is a well-defined center cluster that is connected to every other cluster by a blocked edge, and this cluster has the largest size in the cluster graph. Thus we can naturally think of two cases: P was the center cluster or P was a satellite cluster. We define a ***satellite cluster*** as a cluster that is not the center, but was connected to the center via a blocked edge before the split of P.

If P was a satellite cluster that split into fragments P_1 and P_2, then only those two clusters could violate the blocked invariant. We can restore the invariant by simply fetching and pushing down one ***outbound edge*** incident to both fragments. We define an outbound edge as an edge incident to a cluster that is not a self-loop. If the outbound edge is blocked, then we are done. If it is unblocked, we push it down thus merging the fragment into another cluster that is incident to a blocked edge.

Restoring the Blocked Invariant With Split Center. When P is the center cluster, the blocked invariant may become violated for P_1 and P_2 themselves, as well as many or all satellite clusters. The difficulty in restoring the blocked invariant in this case is that we have to restore the blocked invariant for potentially many satellites without examining too many blocked edges since they cannot be pushed down to charge the work. The structure of this new cluster graph consists of the two centers, P_1 and P_2, and a set of satellite clusters connected by edges to P_1 and/or P_2. There may also be edges between P_1 and P_2 (previously self-loops on P), edges between satellites, and self-loops on any cluster (see example in Figure 5). Here we give a high-level description of how to restore the blocked invariant after a center has been split. We leave a more careful description and analysis to the full paper [36] due to space constraints. Our result is summarized by the following lemma:

Lemma 5.9. *When a cluster X is split in a cluster graph $CG(c)$ that previously maintained Invariant 5.2, the invariant can be restored in $O((k+1)\log n)$ time where k is the total number of edges pushed down by this process.*

The high-level procedure is to sequentially fetch an outbound edge incident to each satellite cluster. We maintain a running total of the sizes of P_1 and P_2 combined with the sizes of their adjacent satellites that have been discovered and sets of the adjacent satellites found. We update these total sizes when the fetched edge connects to one of the center fragments. If it connects to another satellite, the two satellites merge together (the edge may be blocked, but we prove in the full paper [36] that this can only happen once). This continues until either (1) a blocked edge from every satellite has been found, or (2) P_1 or P_2 has found two neighboring satellites that could not merge with it due to the size constraint (e.g. the two edges to the center fragment would be blocked if all previously discovered satellites were merged with their center). In the first case it is easy to restore the blocked invariant by pushing down all but at most two of the edges found to the center fragments. In the second case, assume without loss of generality that this happened with P_1. Then it is certain that P_2 is able to merge with all of its neighboring satellites without violating the size constraint. Thus, we can fetch all of the edges out of P_2 and merge it with all of its neighbors to restore the blocked invariant.

Once the invariant has been restored, the same procedure as used at the original level is used to check the connectivity of P_1 and P_2 and then possibly continue to higher levels.

Analysis. Lemma 5.10 proves the correctness of our deletion algorithm in maintaining the size invariant and the blocked edge invariant, the formal proof of which is described in the full paper [36]. The overall correctness of the algorithm follows from this lemma and the original cluster forest algorithm.

Lemma 5.10. *Invariants 2.1 and 5.2 are preserved by edge deletion.*

Deletions can take $O(\log n)$ work per level that is not charged to an edge being pushed down if a blocked edge is immediately encountered when certifying connectivity. This results in a total of $O(\log^2 n)$ uncharged work. Combining this with Lemma 5.9 yields the following theorem:

Theorem 5.11. *The amortized cost of insert and delete operations is $O(\log^2 n)$ using the blocked cluster-forest data structure.*

6 PARALLELIZING INDIVIDUAL UPDATES

Next, we step towards our goal of a parallel batch-dynamic algorithm by describing how to perform a single update in parallel. Many ideas and primitives developed in this setting are also useful in the batch-dynamic setting (Section 7). The depth of insertion is $O(\log^2 n)$ already, so we focus on deletion.

6.1 Batch-Dynamic Local Trees

We use a modified version of the local trees used in [47] that also supports parallel operations. Additionally, our parallel algorithm requires an operation that returns a prefix of the smallest clusters in the local tree, sorted by size. The interface for the modified local trees is defined as follows:

- **BatchInsert**$(c_1, ..., c_k)$ takes a list of k new clusters and adds them to the local tree.
- **BatchDelete**$(c_1, ..., c_k)$ takes a list of k clusters in the local tree and removes them.
- **GetMaximalPrefix**(s) returns a maximal prefix of the clusters in the local tree sorted by size such that their total size is less than or equal to s.

To support these operations efficiently, we divide the elements in the tree into $O(\log n)$ size classes of geometrically increasing buckets and store a separate weight-balanced tree for the clusters in each size class. In the weight-balanced trees, elements are keyed by the size of the cluster they represent. This approach allows us to efficiently return a prefix of the elements. Since the sizes of all elements are within a constant fraction of each other, the telescoping sum argument of [47] still applies. Additionally, since weight-balanced trees are known to support efficient batch-parallel insertion and deletion [6], our local trees can also support efficient batch insertions and deletions by first semi-sorting by size class and then separately and in parallel batch inserting and deleting elements into/from the weight-balanced tree for each size class. All operations can be done in $O(\log n)$ depth and $O(k \log(1 + n/k))$ work, which we carefully describe and prove in the full paper [36].

6.2 Searching for Edges in Parallel

To achieve low depth in batch updates we need an operation to fetch k level i edges incident to a given cluster in low depth:

- **FetchEdges**(k, C, i) returns k level-i edges incident to a cluster C, or all of the level-i edges if there are less than k.

To implement this operation, we traverse down the nested local tree structure one level at a time, maintaining a set of approximately k clusters that have a level i edge incident to them by following the set bits in the bitmaps. The algorithm is described in detail in the full paper [36]. This yields the following lemma:

Lemma 6.1. *Fetching k level-i edges incident to a given cluster (or all edges if $< k$ exist) in a cluster forest implemented with local trees can be done in $O(\log n)$ depth and $O(k' \log n)$ work where k' is the number of edges returned.*

Now we describe two useful routines to fetch edges in parallel which will be used in the rest of this section. These are (1) fetching all edges incident to a cluster and (2) fetching a single outbound edge incident to a cluster. Note that fetching a single outbound edge is not trivial because it is possible to repeatedly find self-loops when fetching an edge incident to a cluster. The goal is to implement both of these operations in $O(\log^2 n)$ depth and work within a constant factor of the work of their sequential implementations (e.g. $O(k \log n)$ where k is the total number of edges fetched).

To accomplish this, we perform a doubling search where we fetch edges in rounds, doubling the number of edges fetched in each round. In each round, we use the parallel **FetchEdges** operation which fetches k edges incident to a cluster in $O(\log n)$ depth and $O(k \log n)$ work. This process takes $O(\log n)$ rounds of doubling since there are at most $O(n^2)$ edges. After each round, we check if the search is complete. For the first operation, we know it is done if **FetchEdges** returns $< k$ edges. For the second operation, we check all of the edges returned in parallel to see if they are a self-loop. This process takes $O(\log n)$ depth and $O(\log n)$ work per edge. For both operations we have fetched at most twice as many edges as the sequential implementation and thus performed work within a constant factor of the sequential work. Each of the $O(\log n)$ rounds of doubling search takes $O(\log n)$ depth, giving a total depth of $O(\log^2 n)$. This yields the following Lemma:

Lemma 6.2. *Fetching all of the edges incident to a cluster or fetching an outbound edge incident to a cluster can be done in $O(\log^2 n)$ depth and $O(k \log n)$ work where k is the total number of edges fetched.*

6.3 Pushing Down Edges in Parallel

Updating Bitmaps in Parallel. First we describe the simpler problem of updating the bitmaps in the local trees for a batch of edges that are pushed down from the same level. Updating the bitmaps is equivalent to updating a batch of k augmented values in the nested local trees structure and can be implemented using an atomic compare-and-swap (CAS) operation on each bit along the leaf-to-root path that needs to be updated, quitting early if the CAS fails. In total, the batch bitmap update takes $O(\log n)$ depth and $O(k \log(1 + n/k))$ work.

Pushing Down Groups of Edges in Parallel. Next, we describe how to handle pushing edges down in parallel. We provide the following two routines:

- **PushDownGroup**(E) pushes down a set E of k level i edges, where the edges in E are all incident to a common level $(i-1)$ cluster and the total size in level $(i-1)$ clusters containing their endpoints is $\leq 2^{i-1}$ (e.g. all of E can be pushed down).
- **BatchPushDown**(E) given a set E of level i edges, this pushes down as many edges of E as possible until the only remaining ones are blocked. Formally, this operation ensures that every level $(i-1)$ cluster containing an endpoint of any edge in E is incident to a blocked edge or its parent is isolated.

Pushing down a single level i edge e to level $(i-1)$ requires updating the level i and level $(i-1)$ bitmaps, and possibly combining the local trees of the two level $(i-1)$ clusters containing the endpoints of e. In the sequential setting, our method to maintain the blocked invariant and the size invariant after a single edge push was to push down an edge as far as possible until it became blocked.

When pushing down multiple edges in parallel, there are a few challenges that arise. There may be situations where pushing down one edge causes a different previously unblocked edge to become blocked. Also, combining several local trees in parallel is non-trivial.

Our Approach: Reducing to PushDownGroup. Consider pushing down a batch of k edges incident to a common cluster. Updating the bitmaps for this batch of edges can be done efficiently as described at the beginning of this section. Lemma 6.3 proves that when pushing down a group of unblocked edges incident to a common cluster, at most one of the clusters can be non-isolated (i.e. the cluster has multiple children).

Lemma 6.3. *For any set of k level i clusters in the same cluster graph whose combined size is $\leq 2^i$, at most one of the clusters can have multiple children (it is not isolated).*

Proof. Assume that two or more of the clusters are not isolated. Then there are at least two disjoint blocked edges between level $(i-1)$ clusters. Each blocked edge must have $> 2^{i-1}$ size. Having at least two of these, the total size is $> 2^i$ which is a contradiction. □

This means that all but one of the cluster graphs will consist of a single cluster. Given this observation, we avoid the challenge of merging several level i local trees in parallel, and simply need to delete the isolated clusters from the level i cluster graph and insert them into the cluster graph of the level $(i-1)$ cluster that is not isolated, both of which can be done batch-parallel using our batch-dynamic local trees. We include the full description and analysis for **PushDownGroup** in the full paper [36].

Lemma 6.4. *Pushing down a batch of k level i edges incident to a common level $(i-1)$ cluster can be done in $O(\log n)$ depth and $O(k \log(1 + n/k))$ work.*

Given this routine, we can reduce the problem of pushing down an arbitrary batch of edges to multiple calls of **PushDownGroup**. The general strategy is to take a spanning forest of the edges and decompose it into disjoint stars. Then for each star we merge into the center a maximal prefix of the clusters sorted by size that can be merged into the center without violating the size constraint. We give the algorithmic details of **PushDownBatch** in the full paper [36], which proves the following lemma:

Lemma 6.5. *Given a set E of k level i edges, enforcing that every level $(i-1)$ cluster containing an endpoint of any edge in E is incident to a blocked edge or its parent is isolated, can be done in $O(\log^2 n)$ depth and $O(k \log n)$ work.*

6.4 Parallelizing Deletion

Here we describe how to parallelize a single update in the blocked cluster forest using the results of the previous subsections. We focus on deletion since insertion already takes $O(\log^2 n)$ depth.

Pushing Down Edges. During a deletion, the algorithm sweeps up the levels of the cluster forest until a replacement edge is found or the top is reached. At each level some edges may be pushed down. In the sequential case, these edges were immediately pushed down as far as possible. To parallelize deletion, we will collect all of the edges that are pushed down at each level during this upward sweep, and handle them later in a downward sweep to restore the blocked invariant. Let E_ℓ be the set of edges pushed down during the upward sweep of deletion at a level ℓ.

Once the upward sweep has finished at level ℓ_u, we start a downward sweep, starting at level $\ell_d = (\ell_u - 1)$. At each level ℓ_d in the downward sweep, we call **BatchPushDown**(E_{ℓ_d}). Every edge that is pushed down by this call is added to the set $E_{(\ell_d - 1)}$.

Restoring the Blocked Invariant. When a cluster is split at higher levels during deletion, the algorithm must restore the blocked invariant in that cluster graph before continuing. As in the sequential setting, if the split cluster was a satellite, we just need to find and/or push down a single outbound edge incident to both fragment clusters. Lemma 6.2 proves that we can find such an edge in $O(\log^2 n)$ depth and $O(k \log n)$ work which can be charged to the self-loops that are found and pushed down. If the split cluster is the center, restoring the blocked invariant is much more complex. Due to space constraints, we leave the description of this to the full paper [36]. Our result is summarized by the following lemma:

Lemma 6.6. *When a cluster X is split in a cluster graph $CG(c)$ that previously maintained Invariant 5.2, the invariant can be restored in $O(\log^2 n)$ depth and $O((k+1)\log n)$ work where k is the total number of edges pushed down by this process.*

We leave the full analysis of deletion to the full paper [36]. Our result is the following lemma:

Lemma 6.7. *A deletion in a blocked cluster forest can be done in $O(\log^3 n)$ depth and $O(k \log n + \log^2 n)$ work where k is the total number of edges pushed down during the deletion.*

7 PARALLEL BATCH-DYNAMIC UPDATES

In this section we show how to extend blocked cluster forests to support parallel batch-dynamic operations.

7.1 Batch Insertion

Consider a batch E of k edge insertions. All of the edges in the batch are inserted at level $(L_{\max}+1)$. Let the set of edges be $E_{L_{\max}+1}$. Then we call **BatchPushDown** on $E_{L_{\max}+1}$. Let $E_{L_{\max}}$ be the set of edges that were pushed down by this call. We repeatedly call **BatchPushDown** on E_i, the set of edges pushed down by the call on E_{i+1}. Algorithm 1 shows the pseudo-code for batch insertion.

Algorithm 1 BatchInsertion(CF, E)

```
1: Insert all of E into the edge lists of their endpoints
2: Batch update bitmaps for level (L_max + 1) edges
3: while |E| > 0 do
4:     E ← BatchPushDown(E)
```

When a batch of edges is introduced into a level, only the clusters incident to those edges may violate the blocked invariant. Calling **BatchPushDown** ensures that these clusters follow the blocked invariant when it is finished. Doing this for every level ensures that the blocked invariant is maintained throughout the cluster forest. Each call to **BatchPushDown** takes $O(\log^2 n)$ depth, so the total depth of batch insertion across all levels is $O(\log^3 n)$. The work in the first call to **BatchPushDown** is $O(k \log n)$. This work can be charged to the k edges in E being inserted. Each subsequent level i performs $O(|E_i| \log n)$ work which can be charged to the $|E_i|$ edges pushed down at the previous level.

Algorithm 2 BatchDeletion(CF, E)

1: Delete all of E from the edge lists of their endpoints
2: Batch update bitmaps for each edge's level prior to deletion
3: $[E_1 \dots E_{L_{\max}}] \leftarrow$ semisort(E), $\ell \leftarrow 1, A \leftarrow E_1$
4: **while** $|A| > 0$ **do** ▷ sweep up
5: $Groups \leftarrow$ sort A by parent
6: **for** $(Group, P) \in Groups$ **do** ▷ parallel for
7: RestoreBlockedInvariant($CG(P)$)
8: $Components \leftarrow \{\}$
9: **for** $Cluster \in Group$ **do** ▷ parallel for
10: $e \leftarrow Cluster$.FetchOutboundEdge()
11: **if** $\neg e$ **then** $Components$.Insert($Cluster$)
12: **else** $Components$.Insert($null$)
13: **if** $|Components| > 1$ **then**
14: **for** $(CC \neq null) \in Components$ **do** ▷ parallel for
15: Create a new parent cluster for CC
16: P.RemoveChildren($Components$)
17: Parent(P).AddChildren(new parents)
18: $A \leftarrow P \cup$ new parents created
19: $\ell \leftarrow \ell + 1, A \leftarrow A \cup \{a \mid a \in e \in E_\ell\}$
20: **for** $\ell \in [L_{\max} - 1, 1]$ **do** ▷ sweep down
21: $D_{(\ell-1)} \leftarrow D_{(\ell-1)} \cup$ BatchPushDown(D_ℓ)

7.2 Batch Deletion

Consider a batch of k edge deletions. The general strategy will be to keep an ***active set*** A of fragment clusters whose connectivity may have changed. First there will be a bottom-up sweep on the levels at the cluster forest. At every level the algorithm (1) restores the blocked invariant (2) performs connectivity search, and (3) possibly splits clusters in the level above. In this upward sweep, edges will be pushed down only one level which may temporarily violate the blocked invariant in the lower levels that have already been processed. Afterwards, there will be a downward sweep to push down any edges that were not pushed down as far as possible to maintain the blocked invariant. Algorithm 2 shows the pseudo-code for batch deletion.

First all of the edges in the batch are deleted from the list of edges at the leaves of the cluster forest and the bitmaps are updated. The next step is to semisort all of the deleted edges in the batch by their (former) level. During the upward sweep at level i, we will add the level i clusters containing the endpoint of each level $(i+1)$ edge to our set of fragment clusters. When subject to k deletions, clusters at upper levels may be split into as many as $k+1$ fragments now instead of just two. The full paper [36] describes how the blocked invariant can be restored in parallel given the possibility of several clusters being split by the batch of deletions. This yields Lemma 7.1.

Lemma 7.1. *Given a cluster graph $CG(c)$ that was subject to k clusters being split and previously maintained Invariant 5.2, the invariant can be restored in $O(\log^2 n)$ depth and $O((x+k)\log n)$ work where x is the total number of edges pushed down by this process.*

Parallel Connectivity Search and Splitting Clusters. Consider the active set of $O(k)$ clusters at level i. They can be grouped based on their level $(i+1)$ parent cluster, by finding the parent of every cluster in $O(k \log n)$ work, and sorting the clusters by parent in $O(\log k)$ depth and $O(k \log k)$ work (line 5) [12]. Then each group can be processed independently in parallel. Let k' be the number of active clusters in a given group. Within each group, we first fix the blocked invariant in the cluster graph of the parent (line 7). This takes $O(\log^2 n)$ depth and results in $O(k' \log n)$ uncharged work by Lemma 7.1.

Then we want to determine the connected components of the cluster graph $CG(P)$ of the parent P. We will take advantage of the fact there cannot be a matching of size greater than one over the blocked edges in $CG(P)$. Since the blocked invariant has been restored, only one connected component can contain multiple clusters in $CG(P)$. The strategy will be to attempt to find an outbound edge for each cluster in the active set. This can be done in $O(\log^2 n)$ depth with $O(\log n)$ uncharged work per cluster, the rest of the work is charged to the self-loops that were found and pushed down. There will be $O(k' \log n)$ total uncharged work. If an outbound edge is found, it is a part of the component with multiple clusters. Each other component is defined by a single cluster. We produce a set of the connected components, representing each singleton component with its cluster and representing the component with multiple clusters as null if it exists (lines 8–12).

If there is only one component, the deletion is done within this component, meaning every edge deletion in this cluster graph has certified connectivity (line 13). Otherwise, each lone cluster is removed as a child of P, and will get a new parent node at level $(i+1)$ (lines 14–15). These will be added to the modified local tree of the parent of P using **BatchInsert**, and their sizes will be subtracted from $n(P)$ within the **AddChildren** function (line 17). The component with multiple clusters will keep the original P as its parent. If it didn't exist (e.g. $n(P) = 0$ now), we delete P by removing it as a child of its parent. The active set at the next level up will be the set of new level $(i+1)$ clusters and possibly P for any group that still had multiple components, along with the clusters containing endpoints of level $(i+2)$ edge deletions (lines 18–19).

Downward Sweep. Just like in parallelizing a single deletion, we will collect all of the edges that are pushed down at each level during the upward sweep, and handle them later in a downward sweep to restore the blocked invariant. Let D_ℓ be the set of edges pushed down during the upward sweep of deletion at a level ℓ. We start the downward sweep at level $L_{\max} - 1$. At each level ℓ in the downward sweep, we call **BatchPushDown**(D_ℓ) (lines 20–21). For every edge that this call pushes down, we add it to the set $D_{(\ell-1)}$.

Cost Analysis. We analyze the work and depth of our batch update algorithms in the full paper [36], yielding the following theorem:

Theorem 7.2. *Batch insertions and batch deletions of edges in the blocked cluster forest can be done in $O(\log^3 n)$ depth per batch with an amortized work of $O(\log^2 n)$ per edge.*

8 EMPIRICAL EVALUATION

In this section, we provide the first experimental study of the cluster forest algorithm. Our goals in this part of the paper are to understand (1) whether the cluster forest algorithm is practical and yields good query and update performance; (2) whether the theoretical space improvements provided by the cluster forest algorithm translate into meaningful space improvements in practice;

and (3) whether the algorithm can scale to large graphs and work well across a variety of different graph types (both real-world and synthetic). Since there have been no prior implementations of the cluster forest approach, we focus our study on sequential implementations in order to carefully study different design choices in the algorithm and carefully measure the impact of these choices on runtime and query performance. In this section, we demonstrate that a carefully optimized implementation of the cluster forest algorithm can achieve all three of these goals, and that the cluster forest approach may be the algorithm of choice when both theoretical guarantees and practical performance are required.

8.1 CF Algorithm Optimization

One of the main contributions of this paper is the first practical and highly-optimized implementation of the cluster forest algorithm. A major bottleneck when implementing the algorithm is the cost of traversing the cluster-forest hierarchy, a step that occurs in nearly all aspects of the algorithm (e.g., replacement edge search, fetching a level i edge, or pushing an edge from level i to $i-1$). Although the hierarchy has depth $O(\log n)$, traversals can still be costly as they encounter both cluster forest nodes and local tree nodes, and thus every traversal involves significant pointer jumping. To address this issue, we designed our implementation to reduce the cost of and eliminate tree traversals whenever possible. Due to space constraints, we provide a more detailed description and discussion of our optimizations in the full paper [36].

Our first optimization is called ***flattened local trees***. As the name suggests, we flatten the local tree structure into an array that stores the roots of the rank trees in increasing order of rank instead of combining the rank trees into a binary tree (see Section 4 for local and rank tree definitions). Since there are at most $\log_2 n$ rank trees, this array approach does not sacrifice the time complexity of any local tree operation and improves locality. Additionally, we do not combine rank trees nodes unless there are more than $\log_2 n$ of them. This means that in many cases where a node in the cluster forest has few children, the algorithm can avoid a large amount of indirection in traversing the local trees and rank trees.

Our next optimization is called ***lowest common ancestor (LCA) insertion***. The optimization is to insert an edge at the level of the lowest level node in the cluster forest that contains both endpoints of the edge. A faithful implementation of the cluster forest algorithm is to simply perform ***root insertion***, i.e., every non-tree edge insertion is simply placed at the root of its tree. The LCA optimization trades off extra time spent during an insertion to find the LCA to distribute the edges better across the levels to achieve faster deletions, since fewer edges need to be searched, thus lowering the amount of traversals. The performance overhead of performing LCA insertion is negligible compared to root insertion—on average LCA insertion is 1.06× faster on the graphs we evaluated when comparing total insertion time. However, LCA insertion makes a huge difference for speeding up deletions—on average, it speeds up total deletion time by 1.5× on average over root insertion across all of our graphs. We note that the LCA optimization is somewhat unique to the cluster forest algorithm, since finding the LCA can be done in $O(\log n)$ time using the cluster forest representation; applying a similar optimization to HLT seems more complex as it requires quickly finding the lowest level where the endpoints of an edge are connected, which naively takes $O(\log^2 n)$ time.

When evaluating an early version of our algorithm, we found that on dense graphs, our implementation was slower than HLT, even when using the LCA optimization. The reason for HLT's speed is that $\geq m - n + 1$ edges are *non-tree edges*, and therefore, many deletions in dense graphs target non-tree edges. HLT benefits from this fact, since a non-tree edge deletion simply checks a hash-table storing whether an edge is tree or non-tree in $O(1)$ time, and updates bitmaps after deleting the edge. To achieve similar benefits in the cluster forest algorithm, we introduce a **non-tree edge tracking** optimization. We give a full description of the optimization in the full paper [36]. The main idea of the optimization is to carefully mark edges as either tree or non-tree—as in HLT, non-tree edges do not affect the connectivity and can simply be deleted. On dense graphs, the optimization yields a significant speedup—for example, on Orkut, we observed a speedup of 1.87× after implementing the optimization due to 87% of the deletions being detected as non-tree edges. Compared to HLT, which detects 89% of the deletions as non-tree edges our optimization shows that our CF implementation can almost completely match the HLT implementation's ability to avoid unnecessary work during edge deletions.

Table 1: Graph datasets used in our experiments.

Name	Type	$\|V\|$	$\|E\|$	Avg Deg	Cite
GER	Road	12.28M	16.12M	2.62	[40]
USA	Road	23.95M	28.85M	2.41	[40]
HH	K-NN	2.05M	6.50M	6.35	[37]
CHEM	K-NN	4.21M	14.83M	7.05	[19]
YT	Web	1.16M	2.99M	5.16	[49]
POKE	Web	1.63M	22.30M	27.32	[34]
WT	Web	1.79M	25.44M	28.41	[50]
EW	Web	4.21M	91.94M	43.72	[9]
SKIT	AS	1.70M	11.10M	13.08	[33]
SO	Temporal	6.02M	28.18M	9.36	[42]
LJ	Social	4.85M	42.85M	17.68	[4]
ORK	Social	3.07M	117.19M	76.28	[49]
TWIT	Social	41.65M	1.20B	57.74	[32]
FR	Social	65.61M	1.81B	55.06	[49]
GRID	Synthetic	10.00M	10.22M	2.04	[14]
RMAT	Synthetic	67.11M	670.83M	19.99	[10]

8.2 Experimental Setup

All of the experiments presented in this paper were run on a machine with 4 × 2.1 GHz Intel Xeon(R) Platinum 8160 CPUs (each with 33MiB L3 cache) and 1.5TB of main memory.

Implementations. Our cluster forest implementations are all written in C++ and use B-tree sets and flat hash sets from Abseil [3]. We refer to our implementations with and without the LCA insertion optimization as ***CF-LCA*** and ***CF-Root***, respectively. We compare against a faithful implementation of the original HLT algorithm [26] written in C++ also optimized using set data structures from Abseil. Our implementations all use -O3 optimization. We also compare against D-Tree [11], a recently published data structure for dynamic connectivity that is written in Python. We note that since D-Tree

is implemented in Python comparing its running time and memory usage with other implementations written in C++ may be unfair; however, since D-tree is a recent linear-space dynamic connectivity algorithm, we include it for completeness.

Input Data. The graphs used in our experiments are summarized in Table 1. We use a variety of real-world and synthetic graphs with varying sizes, densities, and types. To generate dynamic updates, we generate a random permutation of all of the edges in the graph, and insert all of the edges in this order. Then we generate another random permutation of the edges and delete all of the edges in this order. We break these random permutations into 10 ***stages*** of inserting $|E|/10$ edges per stage and then 10 stages of deleting $|E|/10$ edges. We use stages to understand how memory usage and the insert and delete speeds change over time (e.g., as the graph grows more dense, and then more sparse). At the end of each stage, we perform 1M queries: half the queries are completely random and half are endpoints of edges that exist in the graph at that point.

8.3 Performance Results

Memory Usage. We start by investigating whether the theoretical guarantees on space provided by the CF algorithm translate into practical improvements in memory usage. Our goal is to determine (1) whether memory usage is a limiting factor in the ability of HLT-based implementations to scale to extremely large graphs, (2) whether implementations based on the CF algorithm using linear space can overcome this obstacle, and (3) whether the memory usage of implementations based on the CF algorithm can perform better than existing state-of-the-art dynamic connectivity implementations using linear space. Our results affirmatively answer all three of these questions.

The top of Figure 6 shows the peak memory usage of each algorithm on the various graphs relative to the memory of CF-LCA. We report the unnormalized numbers for peak memory usage in the full paper [36]. Our main finding is that CF implementations consistently require significantly less memory than HLT. ***CF-Root uses 6.2×–19.7× less memory than HLT and CF-LCA uses 5.7×–17.5× less memory than HLT.*** We note that across all the graphs we tested, the memory efficiency of both CF-Root and CF-LCA are very similar, with CF-Root having a slight edge (1.0×–1.2× less memory). This is because with root insertion there are slightly fewer nodes in the cluster forest. On the densest graph we tested (Orkut) our best CF implementation uses 26 bytes/edge while HLT uses 159 bytes/edge. On the sparsest graph we tested (Grid) the CF algorithm uses 294 bytes/edge while HLT uses 5,809 bytes/edge.

These results clearly show the benefits of the CF algorithm over HLT in practice across a wide variety of real-world graphs with different characteristics. For sparse graphs (e.g., USA Roads, StackOverflow, Grid) where m is closer to n, the $O(n+m)$ space usage of CF should beat the $O(n \log n + m)$ space usage of HLT, and this asymptotic difference can be clearly seen in the results. Interestingly, even for dense graphs (e.g. ENWiki, Orkut, Twitter) where $m > n \log_2 n$, CF still uses significantly less memory than HLT. This is because in dynamic connectivity algorithms edges can be stored very space efficiently ($\approx$ 10 bytes/edge) while the space overhead of the tree data structures used in virtually all dynamic connectivity algorithms scales heavily with the number of vertices (at least a hundred bytes per vertex per tree). This supports the conclusion that on real-world graphs (which are typically quite sparse) dynamic connectivity algorithms that require storing a large number of trees over the vertices (like HLT) prevent scaling to very large graphs. The extra $\log n$ factor on the space usage, which may easily be overlooked in theory, *has a massive impact in practice.*

For the third question, we compare our implementations of the CF approach with an existing implementation of D-Tree [11], which uses linear space in theory but sacrifices worst-case theoretical guarantees on update time. Although it is implemented in Python, D-Tree still uses significantly less memory than HLT in all of our experiments. This is again indicative of the large impact of using data structures that have linear total space in practice.

Update Times. The goals of our next experiments are (1) to determine whether CF implementations can match or beat the performance of HLT for updates despite the increased implementation complexity of CF, (2) to compare the update performance of our CF implementations with the performance of existing state-of-the-art dynamic connectivity implementations, and (3) to investigate the impact of the optimizations from Section 8.1 on update speed.

The middle of Figure 6 shows total update time of each algorithm on the various graphs relative to the time for CF-LCA. We report the raw (unnormalized) numbers for updates times in the full paper [36]. Additionally, we include Figure 7 to show an example of how the update performance varies throughout the sequence of updates (per-stage) on the LiveJournal graph.

Our results show that our CF implementations can achieve significantly better performance than HLT for both insertions and deletions for all all graphs in our experiments. ***CF-LCA performs updates 1.4×–6.2× faster than HLT, and CF-Root performs updates 1.5×–3.8× faster than HLT.*** The improvements in update speed for the CF implementations can be attributed to the optimizations we described in Section 8.1. For example, for non-tree edge insertions, both algorithms must simply add the edge to the edge set, and then traverse up to update $O(\log n)$ bitmaps; the flattened local tree optimization decreases the cost of such traversals in the CF implementation resulting in faster insertions. For non-tree edge deletions, the non-tree edge tracking optimization enables our CF implementation to match the performance of HLT. With the optimization, both algorithms simply check the hash-table once, delete the edges from the edge set, and traverse up to update the bitmaps. The flattened local tree optimization once again allows the CF implementations to outperform the HLT implementation due to the lower cost of traversing the hierarchy. CF-LCA performs better than CF-Root for total update time in most cases. This is because while the LCA optimization slows down insertions slightly, deletions become much faster due to fewer edges needing to be inspected. However, CF-Root benefits from being able to find replacement edges quickly in dense, well connected graphs like Twitter.

We also compared our CF implementations with D-Tree, which sacrifices worst-case guarantees but obtains linear space usage. The update performance of D-Tree varies greatly depending on the graph due to its heuristic nature, but the CF algorithms perform updates faster than D-Tree in all of our experiments. We note that updates for D-Tree can take much longer on certain graphs; e.g., D-Tree is 173× slower than CF-LCA on the Household Lines graph.

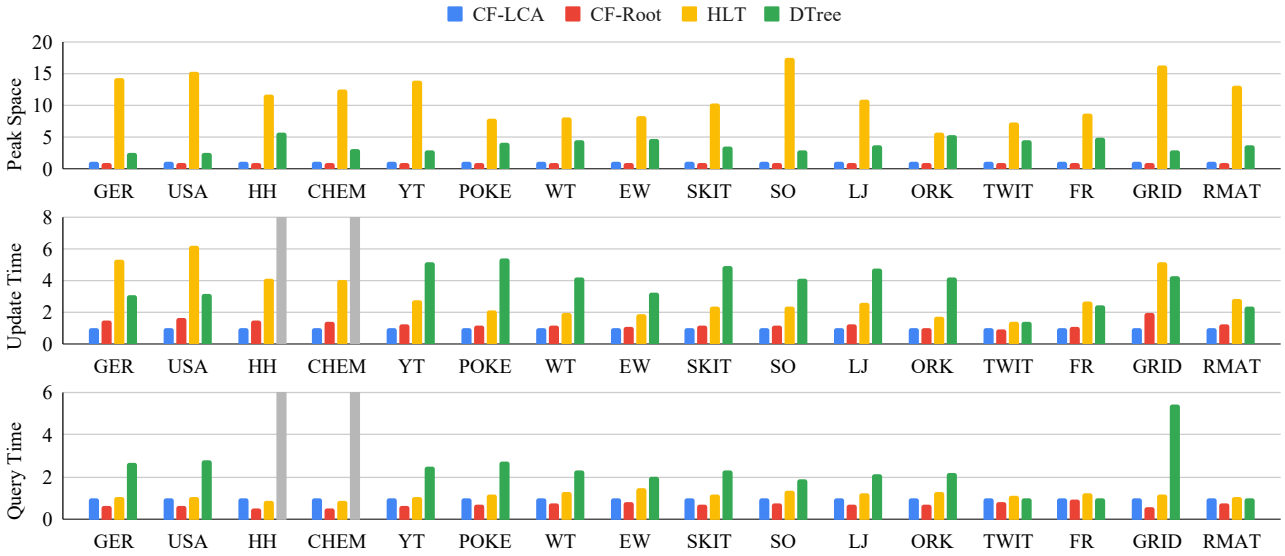

Figure 6: The results of our experiments for peak space usage (top), total update time (middle), and total query time (bottom) of each system on various inputs. All values are normalized to CF-LCA. The un-normalized results are presented in the full paper [36]. A gray bar indicates that the time for D-Tree was over 100× longer than that of CF-LCA on the same input or it terminated before completion with a timeout of 24 hours.

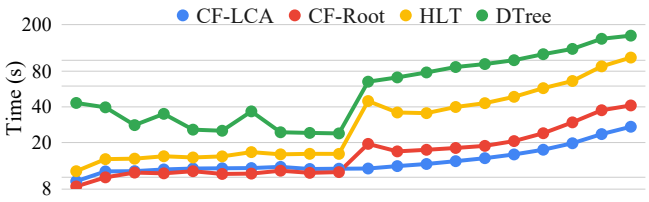

Figure 7: The total time in seconds for each stage of updates on the LiveJournal graph. The points on the x-axis represent the 10 stages of insertion followed by 10 stages of deletion.

Since the algorithm does not have worst-case guarantees, we believe it is unlikely to be much faster on these bad cases even if it was implemented in a different language.

Impact of Graph Properties. Interestingly, graph size does not seem to play a major role in the relative performance of different dynamic connectivity algorithms, and other properties such as graph density, and graph diameter play a more important role. For example, we observe a strong positive correlation between density and improved running time for HLT; e.g., despite Twitter being one of the largest graphs we evaluate on, its high density make HLT perform significantly better.

Query Times. The bottom of Figure 6 shows the total query time of each algorithm on the various graphs normalized to the time for CF-LCA. We report the unnormalized numbers for query time in the full paper [36]. Our main finding is that our CF implementations do not sacrifice any performance in terms of query speed compared to HLT. This is expected because, in both types of algorithms, queries are answered by traversing to the root of the components of the two vertices which requires $O(\log n)$ memory reads and a single equality check. Our results show that queries for our CF implementations are slightly faster than HLT. We believe this is further indicative that the height of the cluster forest hierarchy is generally lower than that of the top level spanning forest data structure in HLT due to the flattened local tree optimization. The query speed for D-Tree varies greatly depending on the graph due to its heuristic nature, but is almost always beaten by the CF algorithms and HLT in our experiments. Comparing CF-LCA and CF-Root, we find that CF-Root always has faster query times because the cluster forest hierarchy has smaller height when performing root insertions.

9 CONCLUSION

This paper makes two significant contributions towards developing a scalable and practical batch-dynamic connectivity algorithm. First, on the theoretical side, we give the first parallel batch-dynamic algorithm for maintaining the connected components of an undirected graph that is work-efficient, runs in polylogarithmic depth, and only uses linear total space. Second, we give the first empirical study of the cluster forest algorithm in the sequential setting, introduce new optimizations to improve its practicality, and demonstrate its superior performance and space-efficiency in practice. Taken together, our results indicate that the CF algorithm is an excellent candidate for a practically scalable dynamic connected components algorithm with good theoretical guarantees.

ACKNOWLEDGMENTS

This work is supported by NSF grants CCF-2403235, CNS-2317194, CCF-1845763, CCF-2316235, CCF-2403237, Google Faculty Research Award, Google Research Scholar Award, and Poland's National Science Centre grant no. 2022/47/D/ST6/02184. We thank the anonymous reviewers for their helpful feedback.

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 (SPAA '19)*. ACM.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (01 Apr 2001).

[3] Abseil C++ Authors. [n.d.]. abseil.io. abseil.io. Online; accessed September 2024.

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Philadelphia, PA, USA) *(KDD '06)*. Association for Computing Machinery, New York, NY, USA, 44–54.

[5] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2019. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *arXiv preprint arXiv:1912.12740* (2019).

[6] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Trans. Parallel Comput.* 9, 2, Article 7 (apr 2022), 41 pages.

[7] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[8] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).

[9] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602.

[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining.. In *SDM*, Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn (Eds.). SIAM, 442–446. http://dblp.uni-trier.de/db/conf/sdm/sdm2004.html#ChakrabartiZF04

[11] 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.

[12] Richard Cole. 1986. Parallel merge sort. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 511–516.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.

[14] Aurelio W. T. de Noronha, André A. Moreira, André P. Vieira, Hans J. Herrmann, José S. Andrade, and Humberto A. Carmona. 2018. Percolation on an isotropically directed lattice. *Phys. Rev. E* 98 (Dec 2018), 062116. Issue 6. https://doi.org/10.1103/PhysRevE.98.062116

[15] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 393–404.

[16] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. 2020. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1300–1319.

[17] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment (PVLDB)* 14, 4 (2020), 653–667.

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

[19] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* (2015).

[20] Kasimir Gabert, Ali Pinar, and Ümit V Çatalyürek. 2021. Shared-memory scalable k-core maintenance on dynamic graphs and hypergraphs. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 998–1007.

[21] Junhao Gan and Yufei Tao. 2017. Dynamic Density Based Clustering. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1493–1507.

[22] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60.

[23] 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.

[24] Monika Rauch Henzinger and Valerie King. 1995. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *ACM Symposium on Theory of Computing (STOC)*. ACM.

[25] Monika R Henzinger and Valerie King. 2001. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. on Computing* 31, 2 (2001), 364–374.

[26] 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.

[27] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. 2017. Fully Dynamic Connectivity in O(Log N(Log Log N)2) Amortized Expected Time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 510–520.

[28] Raj Iyer, David Karger, Hariharan Rahul, and Mikkel Thorup. 2002. An Experimental Study of Polylogarithmic, Fully Dynamic, Connectivity Algorithms. *ACM J. Exp. Algorithmics* 6 (Dec. 2002), 4–es. https://doi.org/10.1145/945394.945398

[29] Bruce M Kapron, Valerie King, and Ben Mountjoy. 2013. Dynamic graph connectivity in polylogarithmic worst case time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[30] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. 2016. Faster Worst Case Deterministic Dynamic Connectivity. In *European Symposium on Algorithms (ESA)*.

[31] Kamran Khan, Saif Ur Rehman, Kamran Aziz, Simon Fong, and Sababady Sarasvady. 2014. DBSCAN: Past, present and future. In *International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*. IEEE, 232–238.

[32] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600.

[33] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations *(KDD '05)*. Association for Computing Machinery, New York, NY, USA, 177–187.

[34] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[35] Quanquan C Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2022. Parallel batch-dynamic algorithms for k-core decomposition and related graph problems. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 191–204.

[36] Quinten De Man, Laxman Dhulipala, Adam Karczmarz, Jakub Łącki, Julian Shun, and Zhongqi Wang. 2024. Towards Scalable and Practical Batch-Dynamic Connectivity. arXiv:2411.11781 [cs.DS] https://arxiv.org/abs/2411.11781

[37] Kolby Nottingham Markelle Kelly, Rachel Longjohn. [n.d.]. *The UCI Machine Learning Repository*. Technical Report. UCI. https://archive.ics.uci.edu

[38] Nicholas Monath, Manzil Zaheer, and Andrew McCallum. 2023. Online Level-wise Hierarchical Clustering. In *ACM Conference on Knowledge Discovery and Data Mining (KDD)*. 1733–1745.

[39] Danupon Nanongkai and Thatchaphol Saranurak. 2017. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and O(n1/2 - ϵ)-time. In *ACM Symposium on Theory of Computing (STOC)*. ACM.

[40] OpenStreetMap contributors. 2017. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org.

[41] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1372–1385. https://doi.org/10.1145/3448016.3457313

[42] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in Temporal Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining* (Cambridge, United Kingdom) *(WSDM '17)*. Association for Computing Machinery, New York, NY, USA, 601–610.

[43] Boyu Ruan, Junhao Gan, Hao Wu, and Anthony Wirth. 2021. Dynamic Structural Clustering on Graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 1491–1503.

[44] Mikkel Thorup. 1999. Decremental dynamic connectivity. *J. Algorithms* 33, 2 (1999), 229–243.

[45] Mikkel Thorup. 2000. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the ACM Symposium on Theory of Computing*. 343–350.

[46] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. 2019. Batch-Parallel Euler Tour Trees. In *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. 92–106.

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

[48] Christian Wulff-Nilsen. 2017. Fully-dynamic minimum spanning forest with improved worst-case update time. In *ACM Symposium on Theory of Computing (STOC)*. ACM.

[49] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (Beijing, China) *(MDS '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages.

[50] Hao Yin, Austin R. Benson, Jure Leskovec, and David F. Gleich. 2017. Local Higher-Order Graph Clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 555–564.