Parallel Batch-Dynamic Minimum Spanning Forest and the Efficiency of Dynamic Agglomerative Graph Clustering

Tom Tseng
MIT CSAIL
Cambridge, Massachusetts, USA
tomtseng@mit.edu

Laxman Dhulipala
University of Maryland
College Park, Maryland, USA
laxman@umd.edu

Julian Shun
MIT CSAIL
Cambridge, Massachusetts, USA
jshun@mit.edu

Abstract

Hierarchical agglomerative clustering (HAC) is a popular algorithm for clustering data, but despite its importance, no dynamic algorithms for HAC with good theoretical guarantees exist. In this paper, we study dynamic HAC on edge-weighted graphs. As single-linkage HAC reduces to computing a minimum spanning forest (MSF), our first result is a parallel batch-dynamic algorithm for maintaining MSFs. On a batch of k edge insertions or deletions, our batch-dynamic MSF algorithm runs in $O(k \log^6 n)$ expected amortized work and $O(\log^4 n)$ span with high probability. It is the first fully dynamic MSF algorithm handling batches of edge updates with polylogarithmic work per update and polylogarithmic span. Using our MSF algorithm, we obtain a parallel batch-dynamic algorithm that can answer queries about single-linkage graph HAC clusters.

Our second result is that dynamic graph HAC is significantly harder for other common linkage functions. For example, assuming the strong exponential time hypothesis, dynamic graph HAC requires $\Omega(n^{1-o(1)})$ work per update or query on a graph with n vertices for complete linkage, weighted average linkage, and average linkage. For complete linkage and weighted average linkage, the bound still holds even for incremental or decremental algorithms and even if we allow poly(n)-approximation. For average linkage, the bound weakens to $\Omega(n^{1/2-o(1)})$ for incremental and decremental algorithms, and the bounds still hold when allowing $n^{o(1)}$-approximation.

CCS Concepts

• **Theory of computation → Shared memory algorithms**; **Dynamic graph algorithms**; • **Information systems → Clustering**.

Keywords

Parallel Algorithms, Dynamic Algorithms, Graph Clustering

ACM Reference Format:
Tom Tseng, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Minimum Spanning Forest and the Efficiency of Dynamic Agglomerative Graph Clustering. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3490148.3538584

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07.
https://doi.org/10.1145/3490148.3538584

1 Introduction

Clustering is a fundamental technique in data mining and unsupervised learning that organizes data into meaningful groups. In this paper, we study *hierarchical agglomerative clustering* (HAC) algorithms. HAC constructs a hierarchy of clusters over a set of points by starting with each point in a separate cluster and merging the two most similar clusters until all points are merged. The similarity between clusters is specified by a *linkage function*. Popular linkage functions include single linkage, complete linkage, average linkage, and weighted average linkage, with average linkage perhaps being the most widely used. Several popular clustering algorithms are based on single linkage as well [7, 26]. HAC on n points can be solved in cubic work in general, and several common linkage functions require only quadratic work [8]. Quadratic work is optimal in the sense that if the input is an $n \times n$ similarity matrix for the n points, then all matrix entries need to be read to compute HAC.

Because the similarity matrix has lots of negligible entries in many scenarios, Dhulipala et al. [18] recently studied *graph-based HAC* (graph HAC) as opposed to the traditional *point-based HAC*. In graph HAC, not all similarities between points need to be specified. Instead, the input is a graph with edges weighted by the similarity between their endpoints. Dhulipala et al. develop exact and approximate algorithms for graph HAC with subquadratic work on sparse graphs and empirically showed that the resulting clusters are of similar quality to those of point-based HAC.

Modern data sets are large and are often rapidly changing, and so a natural question is whether we can compute HAC over a dynamic data set. Even with subquadratic work, it is inefficient to statically re-compute HAC on every update of a large, dynamically changing graph. Little research has been done on dynamic HAC. Graph HAC seems more likely to yield fast dynamic algorithms than point-based HAC—a graph update can be as granular as updating a similarity between one pair of vertices, whereas updating points in point-based HAC incurs $\Omega(n)$ changes in the similarity matrix. As such, this paper aims to study whether graph HAC allows efficient dynamic algorithms under edge insertions and deletions.

The canonical output for HAC is a dendrogram showing the hierarchical clustering, but there are graphs for which one edge update can completely change the structure of the dendrogram. It therefore seems that a dynamic HAC algorithm that explicitly maintains a dendrogram will have poor worst-case update time. We hence examine dynamic graph HAC algorithms with more restricted query outputs, e.g., queries of the form "are query vertices s and t in the same cluster if we agglomeratively cluster until all similarities are below query threshold θ."

With this form of query, single-linkage graph HAC indeed admits efficient dynamic algorithms. As single-linkage HAC reduces

to computing a minimum spanning forest (MSF) [24], we can solve dynamic single-linkage HAC by first applying a dynamic MSF algorithm. The state-of-the-art dynamic sequential MSF algorithm achieves $O(\log^4 n/\log\log n)$ amortized work per edge update to maintain an MSF [30]. Then, storing the MSF in a dynamic trees data structure [45] allows us to answer the queries in logarithmic work. To support a high velocity of updates, however, we may want a *batch-dynamic* algorithm that can batch together updates and exploit parallelism across a batch. Though there are efficient parallel batch-dynamic algorithms for connectivity and incremental MSF [2, 6], no such algorithm has been developed for general dynamic MSF.

This discussion raises two questions: (1) Can we develop a parallel batch-dynamic MSF algorithm, hence giving an parallel batch-dynamic algorithm for single-linkage graph HAC? (2) Do other linkage functions also admit dynamic algorithms with polylogarithmic work per update?

In this paper, we give a parallel batch-dynamic MSF algorithm achieving $O(k\log^6 n)$ expected amortized work and $O(\log^3 n\log k)$ span with high probability (w.h.p.)[1] for a batch of k edge insertions or k edge deletions. Moreover, our MSF result is of independent interest outside the context of clustering. Prior to our algorithm, there was not even a batch-decremental MSF algorithm with polylogarithmic span achieving $O(kn)$ work on edge deletions.

We first give a parallel batch-decremental MSF algorithm achieving $O(\log^4 n)$ expected amortized work per edge and $O(\log^3 n\log k)$ span w.h.p. per batch. A key challenge in parallelizing the decremental MSF algorithm is fetching the k lightest edges incident to a connected component in low span. We solve this approximately by augmenting an internal data structure with quantile summaries. Then, we parallelize Holm et al.'s reduction from sequential decremental MSF to fully dynamic MSF [29] to obtain our batch-dynamic MSF algorithm.

On the other hand, even under our restricted query model for dynamic HAC, we show polynomial conditional lower bounds on the work of dynamic graph HAC for complete linkage, weighted average linkage, and average linkage, even with $n^{o(1)}$-approximation and even when restricted to incremental or decremental algorithms. Table 1 summarizes our lower bounds. Our bounds build on past work showing that several dynamic problems have lower bounds conditional on conjectures like the strong exponential time hypothesis (SETH) [11] via reductions [1, 27].

Our contributions are summarized as follows:

- We parallelize a relative-error quantile summary data structure (see the full version of our paper [49]) and use it to solve parallel batch-decremental MSF in $O(\log^4 n)$ expected amortized work per edge and $O(\log^3 n\log k)$ span w.h.p. per batch of k edge deletions (Section 4.3).
- We use batch-decremental MSF to solve parallel batch-dynamic MSF (and hence also parallel batch-dynamic single-linkage graph HAC) in $O(k\log^6 n)$ expected amortized work and $O(\log^3 n\log k)$ span w.h.p. on a batch of k edge insertions or edge deletions (Section 5). These are the first decremental and fully dynamic MSF algorithms achieving polylogarithmic work per update and polylogarithmic span per batch.
- We prove polynomial conditional work lower bounds for dynamic and partially dynamic graph HAC with complete linkage, weighted average linkage, and average linkage (Section 6). For example, assuming the SETH, dynamic HAC takes $\Omega(n^{1-o(1)})$ per update or query for all of these linkage functions, even with $n^{o(1)}$-approximation.

[1]We say that an event occurs *with high probability (w.h.p.)* if it occurs with probability at least $1-1/n^c$ for any $c\geq 1$, where constants inside asymptotic bounds can depend on $\text{poly}(c)$.

Problem	Work lower bounds			
	Preprocess	Update	Query	Conjecture
HAC (complete or weighted average)	$\text{poly}(n)$	$n^{1-\varepsilon}$	$n^{1-\varepsilon}$	SETH
	$m^{1+\delta-\varepsilon}$	$m^{\delta-\varepsilon}$	$m^{2\delta-\varepsilon}$	Triangle
	$m^{4/3-\varepsilon}$	$m^{\alpha-\varepsilon}$	$m^{2/3-\alpha-\varepsilon}$	3SUM
	$\text{poly}(n)$	$m^{1/2-\varepsilon}$	$m^{1-\varepsilon}$	OMv
#HAC (complete or weighted average)	$\text{poly}(n)$	$n^{1-\varepsilon}$	$n^{1-\varepsilon}$	SETH
	$m^{1+\delta-\varepsilon}$	$m^{\delta-\varepsilon}$	$m^{2\delta-\varepsilon}$	Triangle
	$\text{poly}(n)$	$m^{1/2-\varepsilon}$	$m^{1-\varepsilon}$	OMv
HAC (average)	$\text{poly}(n)$	$n^{1-5c-\varepsilon}$	$n^{1-5c-\varepsilon}$	SETH
	$\text{poly}(n)$	$n^{(1-c)/6-\varepsilon}$	$n^{(1-c)/3-\varepsilon}$	OMv
Dec/Inc HAC (average)	$\text{poly}(n)$	$n^{(1-8c)/2-\varepsilon}$	$n^{(1-8c)/2-\varepsilon}$	SETH
#HAC (average)	$\text{poly}(n)$	$n^{1-c-\varepsilon}$	$n^{1-c-\varepsilon}$	SETH
	$m^{(1+\delta)(1-c)-\varepsilon}$	$m^{\delta(1-c)-\varepsilon}$	$m^{2\delta(1-c)-\varepsilon}$	Triangle
	$\text{poly}(n)$	$m^{(1-c)/2-\varepsilon}$	$m^{1-c-\varepsilon}$	OMv
Dec/Inc #HAC (average)	$\text{poly}(n)$	$n^{1-2c-\varepsilon}$	$n^{1-2c-\varepsilon}$	SETH

Table 1: The table states asymptotic work bounds for dynamic graph HAC such that the listed conjecture (defined in Section 6.1) would be falsified. For problems listed as "HAC", queries answer whether two query vertices are in the same cluster after agglomeratively clustering up to a query similarity threshold, and in "#HAC", queries (given a query similarity threshold) answer with the number of clusters. The bounds allow $O(n^c)$-approximation for a constant $c\geq 0$. In the table, $\varepsilon>0$ is an arbitrarily small constant, $\alpha\in[1/6,1/3]$, and $\delta>0$ is some constant for which triangle detection takes $\Omega(m^{1+\delta-o(1)})$ work. The same bounds also hold for partially dynamic algorithms except for the average-linkage bounds based on SETH; we list SETH-based partially dynamic average-linkage bounds separately as "Dec/Inc." The bounds are amortized for fully dynamic algorithms and worst-case for partially dynamic algorithms.

2 Related work

Graph HAC. We use the definition of graph HAC by Dhulipala et al., who give algorithms solving static graph HAC on m edges and n vertices in $O(m\log n)$ expected work for weighted-average-linkage HAC, $\tilde{O}(n\sqrt{m})$ work for average-linkage HAC, and $O(m\log^2 n)$ work for approximate average-linkage HAC [18]. Older papers have also studied graph HAC but with weaker theoretical guarantees [20, 32]. Another line of work has developed the theoretical foundations of HAC by studying the objective function that it optimizes [14, 16, 39].

Dynamic HAC. There is no prior work on dynamic HAC with good approximation or running time guarantees. Menon et al. give an online approximate algorithm for point-based dynamic HAC [36]. Their algorithm does not have rigorous bounds on approximation quality or worst-case running time. Other online clustering algorithms like Perch [33] and Grinch [38] neither compute the same output as HAC nor approximate HAC in a provably efficient way.

HAC lower bounds. Point-based HAC in Euclidean space is at least as hard as finding the closest pair of points. Karthik and Manurangsi show that, assuming the SETH, closest-pair in dimension $\omega(\text{polylog}(n))$ requires $\Omega(n^{2-o(1)})$ work, and $(1+o(1))$-approximate closest-pair in dimension $\omega(\log n)$ requires $\Omega(n^{1.5-o(1)})$ work. These lower bounds do not apply to graph HAC.

Dynamic MSF. In this paper, we focus on edge insertions and deletions. For sequential dynamic MSF, Holm et al. give an algorithm with $O(\log^4 n)$ amortized work per edge update, which was later improved to $O(\log^4 n/\log\log n)$ amortized update work per update [29, 30]. The best worst-case bound is $O(n^{o(1)})$ work per update [13, 40].

For parallel batched edge updates, Anderson et al. give an incremental MSF algorithm that handles k edge insertions in $O(k\log(1+n/k))$ expected work and $O(\log^2 n)$ span w.h.p. [6]. Other existing algorithms are deterministic but have much higher work bounds. Pawagi and Kaiser give an algorithm handling insertions in $O(kn)$ work and $O(\log n\log k)$ span and deletions in $O(n^2(1+\log^2 k/\log n))$ work and $O(\log n+\log^2 k)$ span [42]. Shen and Liang give an algorithm handling insertions and deletions in $O(n^2)$ work and $O(\log n\log k)$ span [44]. Ferragina and Luccio give an algorithm handling insertions in $O(n\log\log\log n\log(m/n))$ work and $O(\log n)$ span and $k=O(n)$ deletions in $O(kn\log\log\log n\log(m/n))$ work and $O(\log n\log(m/n))$ span [19].

For parallel single edge updates, Kopelowitz et al. give an algorithm running in $O(\sqrt{n}\log n)$ work and $O(\log n)$ span per update [34]. There are also many algorithms for the harder problem of dynamic vertex updates, all of which cost $\Omega(n)$ work per update. We refer the reader to Das and Ferragina's survey for an overview of algorithms for vertex updates as well as for edge updates [15].

3 Preliminaries

Graph HAC. We denote a graph by $G=(V,E)$. Graphs are undirected and simple unless noted otherwise. For weighted graphs, we denote a weight or similarity of an edge $\{x,y\}$ either by writing $w(x,y)$ where $w: E\to\mathbb{R}$ is a weight function or by placing weight $w\in\mathbb{R}$ in a tuple $(\{u,v\},w)$. We often denote $n=|V|$ and $m=|E|$. In our asymptotic bounds, we assume $m=\Omega(n)$. We denote the neighbors of $v\in V$ as $N(v)$. We write $\mathrm{Cut}(X,Y)$ to denote the set of edges between two sets of vertices X and Y.

In graph HAC, we are given a weighted undirected graph and a *linkage function* specifying the similarities between clusters. Each vertex starts in its own cluster, and we compute a hierarchical clustering by repeatedly merging the two most similar clusters, i.e., the clusters connected by the highest-weight edge.

In *single linkage*, the similarity $\mathcal{W}(X,Y)$ between two clusters X and Y is the maximum similarity between two vertices in X and Y, i.e., $\max_{(x,y)\in\mathrm{Cut}(X,Y)} w(x,y)$, whereas *complete linkage* takes the minimum similarity. In *average linkage* (which still applies to weighted graphs), the similarity is $\sum_{(x,y)\in\mathrm{Cut}(X,Y)} w(x,y)/(|X||Y|)$. In *weighted average linkage*, if a cluster Z is formed by merging clusters X and Y, then the similarity between Z and an adjacent cluster U is $(\mathcal{W}(X,U)+\mathcal{W}(Y,U))/2$ if edges (X,U) and (Y,U) both exist and is otherwise the weight of the existing edge.

Parallel model. We use the work-span model with arbitrary forking, a closely related model to the classic CRCW PRAM model [9, 31]. Running time bounds are in terms of *work* and *span* (depth). The work of an algorithm is the total number of instructions, and the span is the length of the longest chain of sequentially dependent instructions. We assume that concurrent reads and writes are supported in $O(1)$ work and span. Procedures can fork other procedure calls to run in parallel and can wait for forked calls to finish.

Parallel primitives. We use several existing parallel primitives in our algorithms. Unordered sets can be stored in parallel dictionaries using linear space and handling batch insertions or deletions of k elements in $O(k)$ work and $O(\log^* k)$ span w.h.p. [23]. Lookup costs $O(1)$ work w.h.p. Ordered sets can be stored in search trees called P-trees [10, 46]. Finding an element by rank or splitting a P-tree of n elements takes $O(\log n)$ work [47]. Inserting or deleting k elements takes $O(k\log(1+n/k))$ work and $O(\log n\log k)$ span [46]. A *semisort*, taking an array of n keyed elements and reordering them so that elements with equal keys are contiguous, can be computed in $O(n)$ expected work and $O(\log n)$ span w.h.p. [25]. A *minimum spanning forest (MSF)* is a spanning forest of minimum weight. An MSF on n vertices and m edges can be statically computed in $O(m)$ expected work and $O(\log n)$ span w.h.p. [43].

4 Parallel decremental MSF

This section will show how to perform parallel batch-decremental MSF (supporting batches of edge deletions), and Section 5 will show how to perform parallel batch-dynamic MSF. We accomplish this by parallelizing the sequential dynamic MSF algorithm by Holm, De Lichtenberg, and Thorup (HDT) that runs in $O(\log^4 n)$ amortized work per update [29]. Their MSF algorithm has three steps: first, they give an algorithm for dynamic connectivity; second, they modify that algorithm into an algorithm for decremental MSF (parallelized in this section); and third, they use decremental MSF to create a fully dynamic MSF algorithm (parallelized in Section 5). Without loss of generality, when discussing MSF, we assume edge weights are unique by breaking ties using lexicographic ordering over edges' endpoints.

The relevance of dynamic MSF to dynamic graph HAC is that single-linkage graph HAC can be solved with Kruskal's algorithm for computing a MSF after negating all edge weights [24]. A complication is that although the canonical output for HAC is a dendrogram, explicitly representing the dendrogram is too expensive for dynamic HAC since an edge update can drastically change the dendrogram's structure (see the full version of our paper [49] for examples). Instead, we implicitly represent the dendrogram by dynamically maintaining the MSF for the clustering. We can then extract information about the single-linkage clustering from the MSF. For instance, suppose that we want to answer the following "group-by-cluster" query, a generalization of the type of query discussed in Section 1: given a query set of k vertices $K\subseteq V$, we want to partition K by the cluster that each vertex would be in if agglomerative clustering were run until all similarities fell below a query similarity threshold θ. We can answer such queries in $O(k\log(1+n/k))$ expected work and $O(\log n)$ span w.h.p. by storing the MSF in a rake-compress (RC) tree, computing a compressed path tree P on the MSF relative to K (Section 5.1 describes RC trees and compressed path trees), removing all edges with similarities below θ from P, and computing connected components on P.

4.1 Background

We first discuss existing algorithms and data structures that our work builds upon.

Euler tour trees. Euler tour trees (ETTs) are a data structure for dynamic forests supporting edge insertion, edge deletion, and connectivity queries in $O(\log n)$ deterministic work [28, 37]. Tseng et

al. introduce a parallel batch-dynamic ETT that internally represents each tree in in the forest as a circular skip list containing the tree's vertices and edges [48]. The ETT can be augmented by a combining function $f : D \times D \to D$, with D being an arbitrary domain. After assigning values from D to vertices and edges, we can maintain the sum of f over each tree (i.e., each connected component) in the forest by having each skip list node store the sum of f over a contiguous subsequence of the sequence represented by the node's skip list. Given an augmentation function f that takes $O(W)$ work and $O(S)$ span to compute, a batch of k insertions, k deletions, or k updates to assigned values for the augmentation takes $O(Wk \log(1+n/k))$ expected work and $O(S \log n)$ span w.h.p. on an n-vertex forest. The randomness in the bounds holds against oblivious adversaries who cannot see heights of skip list elements.

Sequential dynamic connectivity. The HDT connectivity algorithm maintains a graph G of n vertices and supports edge insertion, edge deletion, and connectivity queries. The algorithm maintains $\log n$ *levels*. Each edge is assigned a level, and the algorithm maintains subgraphs $G_1 \subseteq G_2 \subseteq \ldots \subseteq G_{\log n} = G$, where G_i contains all edges of level at least i and has the invariant that each connected component has size at most 2^i. The algorithm also maintains spanning forests $F_1 \subseteq F_2 \subseteq \ldots \subseteq F_{\log n}$, where F_i is a spanning forest of G_i. Connectivity queries are answered in $O(\log n)$ work by storing $F_{\log n}$ in an ETT and querying the ETT. An edge insertion is handled by assigning the edge to level $\log n$ and inserting it into $F_{\log n}$ if the edge's endpoints are not connected by a path.

A deletion of an edge e of level ℓ is handled by deleting it from G_i for all $i \geq \ell$. If e is not in $F_{\log n}$, then we are done. Otherwise, the deletion of e splits a connected component in $F_{\log n}$ in two, and we must search for a *replacement edge* reconnecting the two components. We delete e from F_i for $i \geq \ell$ and conduct our search starting on level ℓ. We look at the smaller of the two connected components formed in F_i by the deletion of e. This connected component has size at most 2^{i-1}, and so pushing this entire component to level $i-1$ would not violate the size invariant. We push all level-i tree edges in the component to level $i-1$. Then, we look at non-tree edges incident to the component one-by-one. If the non-tree edge reconnects the two components, then we have found a replacement edge—we change that edge into a tree edge, and we are done. Otherwise, we amortize the cost of looking at this non-replacement edge by pushing it to level $i-1$. We repeatedly run this search on increasing values of i until a replacement edge is found.

If each F_i is stored in an appropriately augmented ETT, then insertions and deletions cost $O(\log^2 n)$ amortized work since each inserted edge can be pushed down at most $\log n$ levels and it costs $O(\log n)$ work to find and push an edge one level using the ETTs.

Parallel batch-dynamic connectivity. Acar et al. developed a parallel batch-dynamic version of the HDT algorithm [2]. We describe the "non-interleaved" version of their algorithm because we will modify it into a decremental MSF algorithm in Section 4.3. (The interleaved version has better running time bounds, but it seems harder to adapt for decremental MSF.)

The main difference from the original HDT algorithm to discuss is how the batch-parallel algorithm finds replacement edges after deleting a batch of edges. The replacement search begins on the minimum level among the deleted edges. When searching on a level i, the algorithm proceeds in $O(\log n)$ rounds. For every component of size at most most 2^{i-1}, we search for a replacement edge out of that component. To achieve low span, each component performs a doubling search, looking at 2^j incident level-i non-tree edges in parallel for increasing j until finding a replacement edge. We push non-replacement edges to the next level to amortize the cost of examining them. We then compute a spanning tree over the replacement edges, keeping only the replacement edges that are in the spanning tree. We proceed to the next round on each "active" component, i.e., each component that still has incident edges to search and that still has size at most 2^{i-1}. After all the rounds, we repeat at higher levels.

By storing each spanning forest in an ETT with appropriate augmentations, the algorithm can process a batch of k edge updates in $O(k \log^2 n)$ expected amortized work and $O(\log^4 n)$ span w.h.p. We note that the span bound can be tightened to $O(\log^3 n \log k)$. The bound has a $\log n$ term from the $O(\log n)$ rounds per level, but since k deletions creates $O(k)$ active components and the active component count decreases geometrically each round, there are only $O(\log k)$ rounds per level.

Sequential decremental MSF. The HDT decremental MSF algorithm is initialized with a graph G of n vertices and maintains the MSF of G while supporting edge deletions. There are only two changes to the algorithm compared to the sequential HDT connectivity algorithm: we initialize $F_{\log n}$ to be the MSF over G, and when we perform a replacement search out of a component, we find the *lightest* replacement edge by looking at incident non-tree edges in increasing weight. Deletions still cost $O(\log^2 n)$ amortized work.

For correctness, the lightest replacement edge for a deleted edge must have the minimum level among all possible replacement edges. Holm et al. prove that the algorithm maintains a *cycle invariant* implying correctness: in every cycle of G, the maximum-weight edge in the cycle is a non-tree edge and has maximum level in the cycle. This invariant holds so long as whenever we push an incident level-i non-tree edge e of a component to level $i-1$, e is lighter than the lightest replacement edge out of the component and we have already pushed all lighter level-i edges incident to this component.

Relative quantile summaries. Consider a set S that is a subset of a totally ordered universe U. For an element $y \in U$, define the rank of y to be the number of elements in S no greater than y: $\text{rank}(y; S) = |\{x \in S \mid x \leq y\}|$.

For $\varepsilon \in (0, 1)$ and a set S, an ε-approximate relative quantile summary Q is a compressed form of the set that can compute queries of the following form: given a rank $r \in [1, |S|]$ such that $[r(1-\varepsilon), r(1+\varepsilon)]$ contains an integer, return an element y such that $\text{rank}(y; S) \in [r(1-\varepsilon), r(1+\varepsilon)]$. For the remainder of this paper, we use the deterministic, mergeable relative quantile summaries described by Zhang and Wang [50]. The full version of our paper [49] re-derives the construction of the summaries since Zhang and Wang's paper omits several proofs of correctness.

Additionally, we show in the full version of our paper [49] that we can parallelize operations on the quantile summaries, which may be of independent interest. The following lemmas give the relevant operations and bounds.

Lemma 4.1. *Given a set S, if we can look up elements of S by rank in $O(W)$ work, we can construct an ε-approximate summary Q of size $|Q| = O(\log(\varepsilon|S|)/\varepsilon)$ in $O(W \log(\varepsilon|S|)/\varepsilon)$ work and $O(W)$ span.*

Lemma 4.2. *Given an approximate summary Q on set S, we can answer queries in $O(\log|Q|)$ work and can obtain the minimum element of S in constant work.*

Lemma 4.3. *Given an integer $b > 0$ and two ε-approximate summaries Q_1 and Q_2 on non-overlapping sets S_1 and S_2, we can create an $(\varepsilon + 1/b)$-approximate summary Q over $S_1 \cup S_2$ of size $O(b \log(n/b))$ in $O(|Q_1| + |Q_2|)$ work and $O(\log(|Q_1| + |Q_2|))$ span.*

We let $\textsc{Combine}(Q_1, Q_2, b)$ denote the algorithm combining summaries Q_1 and Q_2 with parameter b.

4.2 Finding light replacement edges

Like how the HDT decremental MSF algorithm comes from modifying the HDT dynamic connectivity algorithm, we will obtain a parallel batch-decremental MSF algorithm by modifying Acar et al.'s batch-dynamic connectivity algorithm to search for batches of light replacement edge candidates rather than arbitrary candidates. The primary challenge is searching for the lightest non-tree edges incident on a component efficiently in parallel.

As in Acar et al's connectivity algorithm, for each HDT level $\ell \in [1, \log n]$, we store F_ℓ in an augmented batch-dynamic ETT. Using quantile summaries, we add an additional augmentation, described in the proof of the following theorem, that allows fetching the lightest non-tree edges incident on a component at the cost of increasing the running time of edge insertions and deletions for the ETTs. Section 4.2.1 walks through an example of the additional augmentation.

Theorem 4.4. *Let $\ell \in [1, \log n]$, and for each vertex $v \in V$, let $N_{v,\ell}$ represent the level-ℓ non-tree edges incident to v. Using $O(n \log^2 n)$ space w.h.p., we can support the following operations over F_ℓ, with all work bounds being in expectation and all span bounds being w.h.p.:*

- *inserting or deleting k edges to F_ℓ in $O(k \log^2 n \log(1 + n/k))$ work and $O(\log n \log\log n)$ span,*
- *inserting or deleting k edges to $\{N_{v,\ell}\}_{v \in V}$ in $O(k \log^2 n \log(1+ n/k))$ work and $O(\log n(\log\log n + \log k))$ span,*
- *obtaining the $k(1 \pm 1/2)$ lightest edges in $\bigcup_{v \in C} N_{v,\ell}$ of a connected component C of F_ℓ in $O(k \log n)$ work and $O(\log n)$ span.*

Proof. We store F_ℓ in batch-dynamic ETTs. We will augment each ETT skip list node with a quantile summary Q and an integer $t \geq 0$, where Q summarizes the weights of non-tree edges incident on the vertices in the node's subsequence and t indicates the error Q has accumulated from $\textsc{Combine}(\cdot, \cdot, \cdot)$ operations.

The augmented value for a vertex v is given by constructing a $1/4$-approximate quantile summary Q over $N_{v,\ell}$ and setting $t = 0$. Each vertex v stores $N_{v,\ell}$ in an ordered set. A weight-w edge in $N_{v,\ell}$ to a neighbor u is stored as a tuple (w, v, u), and ordering is lexicographic. By Lemma 4.1, using a P-tree to represent the ordered set, we can construct Q in $O(\log^2 n)$ work and $O(\log n)$ span.

Define $b(t) = 8(\log n + t^2/\log n)$ and define the ETT augmentation function f as

$$f((Q_1, t_1), (Q_2, t_2)) = (\textsc{Combine}(Q_1, Q_2, b(t)), t)$$
$$\text{where } t = \max\{t_1, t_2\} + 1.$$

For a skip list node in the ETT whose subsequence has vertices S, its augmented value Q summarizes the weights of $\bigcup_{v \in S} N_{v,\ell}$, and its t is bounded by the longest search path length from that node to a node representing some $v \in S$ at the bottom level of the skip list. Since the maximum path length in a skip list is $O(\log n)$ w.h.p. [17], we have $t = O(\log n)$ and $b(t) = O(\log n)$ for every summary w.h.p. By Lemma 4.3, the augmentation takes $O(\log^2 n)$ space per skip list node and runs in $O(\log^2 n)$ work and $O(\log\log n)$ span, all w.h.p.

Recall that given an augmentation function that costs $O(W)$ work and $O(S)$ span, a batch of k updates to an ETT takes $O(Wk \log(1+ n/k))$ expected work and $O(S \log n)$ span w.h.p. Therefore, with our augmentation function, a batch of k edge insertions or deletions to F_ℓ takes $O(k \log^2 n \log(1+n/k))$ expected work and $O(\log n \log\log n)$ span w.h.p. The cost of insertions or deletions to $\{N_{v,\ell}\}_{v \in V}$ also incurs the same cost in updating augmented values, but there is the additional cost of having to actually update $\{N_{v,\ell}\}_{v \in V}$ and to rebuild the quantile summaries over $\{N_{v,\ell}\}_{v \in V}$.

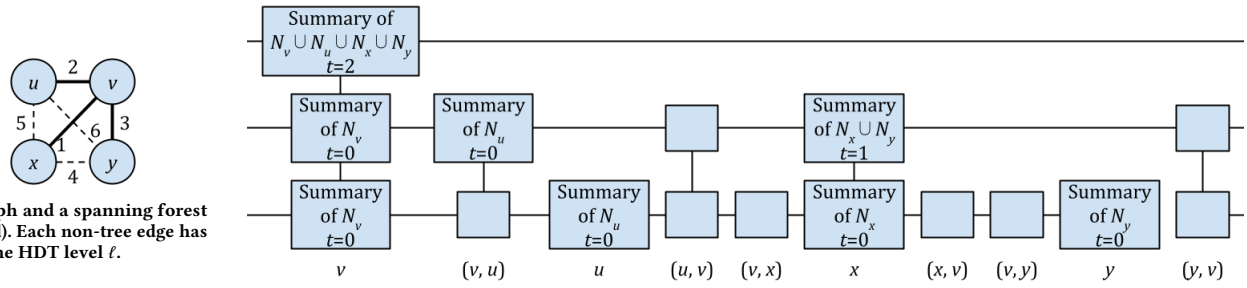
To update $\{N_{v,\ell}\}_{v \in V}$ with k edges U, apply a semisort to group the edges by endpoint: let r be the number of distinct endpoints in U, and let $K = \{(v_1, E_1), \ldots, (v_k, E_r)\}$ represent the semisorted updates, where $N_{v_i,\ell}$ should be updated with edges E_i for each integer $i \in [1, r]$. Updating the ordered set for $N_{v_i,\ell}$ costs $O(|E_i| \log(1 + |N_{v_i,\ell}|/|E_i|))$ work and $O(\log n \log k)$ span. The sum of this work over all v_i is $O(k \log n)$. We then rebuild the quantile summary for each v_i from scratch via Lemma 4.1 in $O(r \log^2 n)$ total work and $O(\log n)$ span. Adding the cost of updating augmented values gives a total expected work is $O(k \log^2 n \log(1 + n/k))$ and the total span is $O(\log n(\log\log n + \log k))$ w.h.p.

The approximation error of a summary Q in the ETT is bounded above by

$$\frac{1}{4} + \sum_{t=1}^{\infty} \frac{1}{b(t)} = \frac{1}{4} + \sum_{t=1}^{\log n} \frac{1}{b(t)} + \sum_{t=\log n+1}^{\infty} \frac{1}{b(t)}$$
$$< \frac{1}{4} + \sum_{t=1}^{\log n} \frac{1}{8 \log n} + \sum_{t=\log n+1}^{\infty} \frac{1}{8t^2/\log n}$$
$$= \frac{1}{4} + \frac{1}{8} + \frac{\log n}{8} \sum_{t=\log n+1}^{\infty} \frac{1}{t^2} < \frac{3}{8} + \frac{\log n}{8} \int_{t=\log n}^{\infty} \frac{1}{t^2}\, dt = \frac{1}{2}.$$

That is, Q is always a $1/2$-approximate quantile summary. We can therefore fetch the $k(1 \pm 1/2)$ lightest edges of $\bigcup_{v \in C} N_{v,\ell}$ for a connected component C by querying the summary of component C for a weight w whose rank is k. Then, by checking whether the summaries' minimum element is less than w, we traverse down the skip list to efficiently find all vertices v in C such that $N_{v,\ell}$ has edges lighter than w in $O(k \log(1 + n/k))$ expected work and $O(\log n)$ span w.h.p. We fetch those edges from each vertex v by splitting the ordered set for $N_{v,\ell}$ in $O(\log n)$ work and $O(\log n)$ span per vertex for $O(k \log n)$ total work. □

4.2.1 Example of augmentation. This subsection gives an example illustrating the ETT augmentation from Theorem 4.4. Figure 1 displays an example graph $G = (V, E)$ with one connected component, a spanning tree for the graph, and a possible skip list internally held by an ETT representing the tree. For simplicity, every non-tree edge



(a) Graph and a spanning forest (in bold). Each non-tree edge has the same HDT level ℓ.

(b) Skip list of the ETT for the graph.

Figure 1: The figure displays a graph and a possible height-3 skip list corresponding to an ETT representing the spanning forest of the graph. The skip list is circular, so the last node at each skip list level is connected to the first node at the same level. The labels below the skip list nodes at the bottom skip list level denote the vertex or edge that the node represents. At higher levels, the skip list nodes represent a contiguous subsequence of the overall sequence (e.g., the fourth node at the middle level represents $(x, (x, v), (v, y), y)$, and the node at the top level represents the whole sequence). Following the augmentation of Theorem 4.4, each list node is augmented with a quantile summary for the vertices within its subsequence. In the figure, the labels inside each skip list node are the augmented values for each node. Some nodes contain no label because their subsequence contains no vertices.

in the graph has the same HDT level ℓ, and we write N_v instead of $N_{v,\ell}$ to denote the non-tree edges incident on a vertex v.

The skip list contains a sequence given by taking an Euler tour on the spanning tree after duplicating each edge in both directions and after adding a self-loop edge on every vertex. At the bottom level of the skip list, each node representing some vertex $v \in V$ (i.e., representing the self-loop edge for vertex v) constructs a quantile summary over v's incident non-tree edges N_v. For instance, $N_u = \{(5, u, x), (6, u, y)\}$, and the skip list node for u is augmented with a 1/4-approximate quantile summary on N_u.

The quantile summaries at higher levels of the skip list are computed by calling the augmentation function f to combine quantile summaries at the level below, ignoring skip list nodes that correspond to edges and hence have no quantile summaries. For instance, the node at the top level of the skip list computes its quantile summary over $N_v \cup N_u \cup N_x \cup N_y$ by calling f on the quantile summaries for N_v and N_u and then calling f on the result of the previous call along with the quantile summary for $N_x \cup N_y$. Each quantile summary has an associated t value such that the summary is $(1/4 + \sum_{i=1}^{t} 1/b(i))$-approximate, where $b(\cdot)$ is defined in the proof of Theorem 4.4. The $\sum_{i=1}^{t} 1/b(i)$ term is accumulated from Combine calls used to compute f. When taking a union of incident non-tree edges such as $N_x \cup N_y$, we may have an edge twice, once in each direction, e.g., $N_x \cup N_y = \{(4, x, y), (4, y, x), (5, x, u), (6, y, u)\}$ has the weight-4 edge $\{x, y\}$ twice. For our use case of decremental MSF (Section 4.3), the edge duplication does not affect correctness or running time complexity.

If there were multiple nodes at the top level of the skip list, we would also call f to combine all of the quantile summaries at the top level to obtain a quantile summary for the entire connected component. At an arbitrary top-level node, we would store a pointer to that quantile summary so that we can quickly fetch a quantile summary for this connected component.

As an example of updating non-tree edges, suppose we wanted to delete an element from N_y. We perform the deletion on N_y, rebuild the quantile summary for N_y entirely from scratch, and then apply f again to rebuild the quantile summaries for the fourth node at the middle skip list level and for the node at the top level.

As an example of searching for light non-tree edges, suppose that we wanted to find the lightest $k(1 \pm 1/2)$ non-tree edges incident on this component for some k. Define the *children* of a skip list node with associated subsequence S to be the nodes at the level immediately below whose subsequences constitute S, e.g., the children of the fourth node at the middle level are the sixth through ninth (inclusive) nodes at the bottom level. We first query the quantile summary at the top level of the skip list for a rank-k entry. Suppose that it returned $(4, y, x)$. Then, we traverse down to the middle level to inspect the quantile summaries of the node's children. At every quantile summary whose minimum element is at most $(4, y, x)$ lexicographically, we traverse down to *that* node's children. In this example, only the fourth node at the middle level satisfies this condition. We again check childrens' quantile summaries' minimum elements, and in this case, the sixth and ninth nodes at the bottom level satisfy the condition. Since we have reached the bottom of the list, we directly access N_x and N_y and fetch all elements that are at most $(4, y, x)$.

4.3 Parallel batch-decremental MSF

As with the sequential HDT decremental MSF algorithm, two changes are needed to change Acar et al.'s batch-dynamic connectivity algorithm into a batch-decremental MSF algorithm. First, given an input graph G, we compute an MSF F over G and set $F_{\log n} = F$. The MSF for the graph will always be $F_{\log n}$.

Second, when performing a doubling search out of a component to find a replacement edge, instead of looking for 2^j arbitrary incident non-tree edges on phase j of a doubling search, we use the ETT augmentation from Theorem 4.4 to search for the $2^j(1 \pm 1/2)$ lightest incident non-tree edges. To maintain the HDT cycle invariant, we do not push any edges on a doubling phase in which we find a replacement edge. In addition, to reduce span, we defer pushing edges to the end of the entire replacement search on a level rather than pushing non-tree edges after every doubling phase.

Theorem 4.5. *We can initialize a batch-decremental MSF data structure in $O(\log^2 n)$ span w.h.p. The data structure supports batches of k edge deletions in $O(\log^3 n \log k)$ span w.h.p. and uses $O(m + n \log^3 n)$ space w.h.p. The total expected work across initialization and all deletions is $O(m \log^4 n)$, i.e., $O(\log^4 n)$ amortized per edge.*

Proof. For initialization, computing $F_{\log n}$ costs $O(m)$ expected work and $O(\log n)$ span w.h.p. Then, by Theorem 4.4, storing $F_{\log n}$ in an augmented ETT and updating the ETT with $O(m)$ incident non-tree edges costs $O(m \log^2 n)$ expected work and $O(\log^2 n)$ span w.h.p.

The span to delete a batch of k edges remains the same as Acar et al.'s algorithm. Despite the increase in span that our more complicated ETT augmentation incurs for insertions and pushing edges, the span is still dominated by the doubling search, whose span remains the same. On the other hand, the work increases by a factor of $O(\log^2 n)$ to a total of $O(\log^4 n)$ expected amortized work per edge due to the increased work for ETT insertion and pushing. The $(1 \pm 1/2)$ uncertainty in searching for incident non-tree edges may increase the amount of amortized cost to charge to each edge by a constant factor, but this does not affect the asymptotic bounds. Not pushing edges found on the last phase of a doubling search also only affects amortized costs by a constant factor.

The maximum space usage is $O(n \log^2 n)$ w.h.p. for each of the $\log n$ ETTs plus $O(m)$ total space to store the non-tree edges. □

5 Parallel fully dynamic MSF

In this section, we describe a parallel batch-dynamic MSF algorithm supporting both batch insertions and batch deletions of edges (Section 5.2) and provide an example execution of the algorithm (Section 5.3).

5.1 Background

Compressed path trees. Like ETTs, *rake-compress (RC) trees* are a parallelizable data structure for dynamic forests [3, 4]. Inserting or deleting k edges from an RC tree takes $O(k \log(1 + n/k))$ expected work and $O(\log^2 n)$ span w.h.p.

Anderson et al. showed that if a dynamic forest F of n vertices is stored in an RC tree, then given k vertices in F ("marked" vertices), we can construct a compressed form of F called a *compressed path tree* relative to the vertices in $O(k \log(1 + n/k))$ expected work and $O(\log n)$ span w.h.p. [6]. The compressed path tree is a forest F' on $O(k)$ vertices (including all marked vertices) such that the heaviest edge on the path between any pair of marked vertices has the same weight in F' as in F. More specifically, the compressed path tree is the union of the paths between the marked vertices with all non-marked vertices of degree below three spliced out. Given k edges in F, we can also find which compressed edges in F' correspond to those edges in $O(k \log(1 + n/k))$ expected work and $O(\log n)$ span w.h.p. by traversing up the RC tree that generated F'.

Dynamic MSF. The fully dynamic HDT MSF algorithm supports both insertions and deletions with $O(\log^4 n)$ amortized work per update. In describing this algorithm, we follow the presentation of Holm, Rotenberg, and Wulff–Nilsen [30].

Along with maintaining the MSF F of the graph G (the *global* tree and graph), the algorithm maintains $2 \log n + 1$ subgraphs A_0, $A_1, \ldots, A_{2 \log n} \subseteq G$ and MSFs $F_0, \ldots, F_{2 \log n}$ of each subgraph (the *local* graphs and trees). Each A_i has at most 2^i non-tree edges $A_i \setminus F_i$ (the *edge-count invariant*), and each non-tree edge in G is a non-tree edge of some A_i (the *non-tree-edge invariant*). We maintain decremental MSF data structures over each local graph.

To insert an edge $e = \{u, v\}$, we query for the heaviest edge e' on the path between u and v in F by storing F in a top tree [5]. We replace e' with e in F if e is lighter. Either e or e' now becomes a new non-tree edge. To make the non-tree-edge invariant hold for the edge, we call Update, a subroutine that we describe shortly below, on the edge to insert it as a local non-tree edge.

To delete an edge e, we delete e from all local graphs and obtain a set of $O(\log n)$ local replacement edges R. If e is in F, we delete it from F and need a global replacement edge. Due to the non-tree-edge invariant, the lightest edge r in R reconnecting F is the global replacement edge. We insert that edge into the global tree. Since edges in R (besides r) are global non-tree edges that might now violate the non-tree edge invariant, we call Update on R.

The Update subroutine with input U inserts the edges in U as local non-tree edges. It re-initializes A_j to be $F \cup U \cup \bigcup_{i \le j}(A_i \setminus F_i)$, with j being the minimal value such that this reinitialization respects the edge-count invariant. The new local tree edges for A_j are the edges in F, and the other edges become local non-tree edges. The subroutine then clears A_i for all $i < j$.

The number of tree edges in each A_i may be large, and so we only store them in compressed form. When initializing A_i, we use a top tree to efficiently compute a structure similar to a compressed path tree. Initializing and storing A_i then takes only $O(2^i \log n)$ work and $O(2^i)$ space, and initializing a decremental MSF over A_i costs $O(2^i \log^2 2^i)$ amortized work.

To analyze the work, in Update, the choice of j means that there are at least 2^{j-1} non-tree local edges $U \cup \bigcup_{i<j}(A_i \setminus F_i)$ being pushed up to A_j. These edges pay for the initialization cost of A_j. A non-tree edge costs $O(\log^3 n)$ across its lifetime since it can be pushed up $2 \log n$ times and may pay $O(\log^2 n)$ amortized work on each push to pay for the cost per edge in the newly initialized decremental MSF data structure. Since each global deletion introduces $O(\log n)$ non-tree local edges, the amortized cost of a deletion in the dynamic MSF algorithm is $O(\log^4 n)$.

5.2 Parallel batch-dynamic MSF

Our parallel batch-dynamic MSF algorithm comes from parallelizing the fully dynamic HDT MSF algorithm. The main changes are to use our decremental MSF algorithm from Section 4.3 and to use RC trees instead of top trees for compressing local graphs and for efficient batch insertion.

Algorithm 1 The algorithm that sets global variables to initialize the batch-dynamic MSF data structure on an n-vertex graph.

```
1: procedure Initialize(n)
2:     F ← RC tree on an empty n-vertex graph    ▷ The MSF, i.e., the global tree.
3:     for i = 0, 1, 2, ..., 2 log n do in parallel
4:         A_i ← ∅    ▷ Decremental MSF data structure for the i-th local graph.
5:         T_i ← RC tree on an empty n-vertex graph
6:         (B_{D,i}, B_{I,i}) ← (∅, ∅)
```

Algorithm 1 initializes the data structure on an n-vertex graph. All variables (and only these variables) defined in this algorithm are globally visible. We assume the input graph begins with no edges since input edges can be added separately via batch insertion. The RC tree F maintains the MSF (global tree) (Line 2).

Each A_i is a batch-decremental MSF data structure over the i-th local graph, which is initially empty (Line 4). Whenever we initialize A_i, we will need to compress its tree edges by computing a compressed path tree on the tree edges relative to A_i's non-tree edges' endpoints. The RC tree used to compute the compressed path tree should remain unmodified until A_i's next initialization so that when deleting edges, we can use the RC tree to look up the compressed representations of the edges in A_i. The RC tree T_i serves this purpose for A_i (Line 5). Its value matches A_i's (uncompressed) tree edges at A_i's latest initialization, or equivalently, the value

of F at A_i's latest initialization. To update T_i to match F at A_i's next initialization, we keep buffers $B_{D,i}$ and $B_{I,i}$ representing the difference between T_i and F (Line 6). In particular, $(T_i \setminus B_{D,i}) \cup B_{I,i} = F$. Section 5.3 illustrates an example of how A_i and T_i changes over several edge updates.

Algorithm 2 A helper algorithm for restoring the HDT non-tree-edge invariant.

```
1:  procedure Update(U = {({u_1, v_1}, w_1), ..., ({u_k, v_k}, w_k)})
2:      for i = 0, 1, 2, ..., 2 log n do
3:          U ← U ∪ (NontreeEdges(A_i) \ F)
4:          A_i ← ∅
5:          if |U| ≤ 2^i then
6:              T_i.Delete(B_{D,i})
7:              T_i.Insert(B_{I,i})
8:              (B_{D,i}, B_{I,i}) ← (∅, ∅)
                ▷ Given an RC tree T representing a forest,
                  T.CompressedPathTree(·) takes a list of vertices L and
                  returns a compressed path tree for the forest relative to L.
9:              P ← T_i.CompressedPathTree(⋃_{({u,v},w)∈U} {u, v})
10:             A_i ← Batch-decremental MSF on P ∪ U
11:             break
```

Before discussing batch insertion, we describe the helper function Update (Algorithm 2) that takes non-tree edges U and inserts them in a local graph to satisfy the HDT non-tree-edge invariant. We iterate through each local graph A_i sequentially to find some A_j to re-initialize with U such that the HDT edge-count invariant still holds (Line 2). As we iterate through increasing i, we collapse the non-tree edges of A_i into U (Lines 3 to 4) since pushing them up to level j will pay for the re-initialization cost of A_j. We discard the tree edges of A_i since they are irrelevant to the non-tree-edge invariant. Once we find the level j (Line 5), we update the RC tree T_j to match the global tree F using buffers $B_{D,j}$ and $B_{I,j}$ and clear the buffers (Lines 6 to 8). Then, we use T_j to compress F into a compressed path tree P relative to U so that $O(|P|) = O(2^j)$, and we set A_j to be a newly initialized decremental MSF data structure over $P \cup U$ (Lines 9 to 10). (The compressed local graphs may be non-simple because between a pair of vertices, there can be both one compressed tree edge and one non-tree edge. The decremental MSF algorithm still works in this setting.)

Algorithm 3 The algorithm for inserting a batch of edges.

```
1:  procedure BatchInsert(U = {({u_1, v_1}, w_1), ..., ({u_k, v_k}, w_k)})
2:      P ← F.CompressedPathTree(⋃_{i=1}^{k} {u_i, v_i})
3:      M ← MSF(P ∪ U)
4:      (D, I) ← (P \ M, U ∩ M)
5:      F.Delete(D)
6:      F.Insert(I)
7:      for i = 0, 1, 2, ..., 2 log n do in parallel
8:          (B_{D,i}, B_{I,i}) ← (B_{D,i} ∪ (D \ B_{I,i}), (B_{I,i} \ D) ∪ I)
9:      Update(D ∪ (U \ M))
```

Algorithm 3 gives pseudocode for batch insertion. We start by compressing the global tree F into a compressed path tree P relative to inserted edges U (Line 2). Each compressed edge e in the compressed path tree also stores a pointer to the heaviest edge in the path that e represents in F. In this way we can, for brevity, refer to edges from the compressed path tree and the corresponding heavy edges in the uncompressed tree F interchangeably in the pseudocode. We compute an MSF M over $P \cup U$ (Line 3). Using M, we can determine which edges I from U to insert into the global tree F and which edges D from F get replaced by I (Line 4). We delete D from F, insert I into F, and update the buffers for every local graph A_i (Lines 5 to 8). Finally, we call Update on edges $D \cup (U \setminus M)$ since they are new global non-tree edges that may violate the non-tree-edge invariant (Line 9).

Algorithm 4 The algorithm for deleting a batch of edges.

```
1:  procedure BatchDelete(U = {{u_1, v_1}, ..., {u_k, v_k}})
2:      D ← F ∩ U
3:      F.Delete(D)
4:      for i = 0, 1, 2, ..., 2 log n do in parallel
5:          (B_{D,i}, B_{I,i}) ← (B_{D,i} ∪ (D \ B_{I,i}), B_{I,i} \ D)
6:          U' ← Representation of U in compressed A_i
            ▷ A_i.Delete(·) takes a list of edges, deletes them from A_i, and returns
              the replacement edges used to reconnect A_i.
7:          R_i ← A_i.Delete(U')
8:      BatchInsert(⋃_{i=0}^{2 log n} R_i)
```

Algorithm 4 describes batch deletion. First, we delete the input edges U from the global tree F and update the buffers for every local graph A_i accordingly (Lines 2 to 5). In parallel over every A_i, we want to delete U from A_i, though this requires some effort since A_i is in compressed form. To map each edge e in U to its representation in A_i (Line 6), there are three cases: e appears in compressed form in A_i (because it was a tree edge in A_i when A_i was last initialized), e appears in uncompressed form in A_i, or it does not exist in A_i. To handle the first case, we try looking up e in T_i to get a compressed edge and then try looking that up in A_i. Simultaneously, we try looking up e directly in A_i to handle the second case. If the two cases fail, then we are in the third case and ignore the edge.

Now we can delete U from A_i and extract the local replacement edges R_i that the decremental MSF data structure uses to replace U (Line 7). Finally, we insert the replacement edges into the global tree by calling BatchInsert (Line 8). Although these replacement edges are already global edges, calling BatchInsert has the correct behavior of reconnecting F and calling Update.

Theorem 5.1. *Our dynamic MSF algorithm maintains an MSF in $O(k \log^6 n)$ expected amortized work for a batch of k edge insertions or k edge deletions. Insertions take $O(\log^2 n)$ span w.h.p., and deletions take $O(\log^3 n \log k)$ span w.h.p. The maximum amount of space the data structure uses is $O(m + \min\{m, n \log n\} \log^3 n)$ w.h.p., where m is the maximum number of edges in the graph.*

Proof. *Work*: We first analyze the work for Update (Algorithm 2). We will give $(2 \log n - i) \cdot O(\log^4 n)$ amortization credits to non-tree edges in local graph A_i and get $O(k \log^5 n)$ expected amortized work for Update as a consequence.

Suppose we call Update (Algorithm 2) with k input edges. Let $\bar{U}$ be the original input to Update, and let U_i indicate the value of U after the i-th iteration of Line 3. We give $O(\log^5 n)$ credits to each edge in $\bar{U}$. Let j be the value of i that satisfies the condition on Line 5. The work done by insertions to U (Line 3) across all iterations sums to $O(|\bar{U}| + 2^j)$. Updating T_j on Lines 6 to 7 costs $O((|B_{D,j}| + |B_{I,j}|) \log n)$ expected work, which we charge to the BatchInsert and BatchDelete calls that inserted these elements into $B_{D,j}$ and $B_{I,j}$. Computing the compressed path tree (Line 9) costs $O(|U_j| \log n)$ expected work, and initializing the decremental MSF data structure (Line 10) costs $O(|U_j| \log^4 n)$ expected amortized work. We know that $|U_j| \le 2^j$ and $|U_{j-1}| > 2^{j-1}$ due to the choice of j. We pay for the $O(2^j + |U_j| \log^4 n) = O(2^j \log^4 n)$ work of Update by charging $O(\log^4 n)$ credits to the elements in $|U_{j-1}|$

that we have pushed up to local graph A_j. The remaining expected amortized work is $O(k \log^5 n)$ from the credits we gave to $\bar{U}$.

Batch insertion (Algorithm 3) also costs $O(k \log^5 n)$ amortized expected work due to its work being dominated by Update (Line 9). For instance, RC tree operations and computing an MSF takes only $O(k \log(1 + n/k))$ expected work on Lines 2 to 6. Updating buffers $B_{i,*}$ on Line 8 costs only $O(k \log^2 n)$ amortized total work, where one $\log n$ factor comes from summing over i and the other $\log n$ factor pays for the cost of updating T_i with $B_{i,*}$ in Update.

Batch deletion (Algorithm 4) costs $O(k \log^6 n)$ amortized expected work. Like with batch insertions, the expected work of RC tree operations and dictionary operations is $O(k \log^2 n)$ on Lines 2 to 6. We charge the cost of deleting k edges from each A_i (Line 7) to the initialization of A_i. Finally, we call BatchInsert on $\bigcup_{i=0}^{2 \log n} R_i$ (Line 8) where $|R_i| \leq k$ for $O(k \log^6 n)$ expected amortized work.

Span: The span of Update (Algorithm 2) is $O(\log^2 n)$ w.h.p. Collapsing A_i into U on Line 3 summed over all $O(\log n)$ iterations takes $O(\log n \log^* n)$ span w.h.p. using a parallel dictionary. Then the operations on T_i and A_i in Lines 6 to 10 all take $O(\log^2 n)$ span w.h.p. and are only performed on one value of i.

The span of Insert (Algorithm 3) is also $O(\log^2 n)$ w.h.p. The span is dominated by the RC tree updates (Lines 5 to 6) and Update (Line 9), each of which take $O(\log^2 n)$ span w.h.p. The span of Delete (Algorithm 4) is $O(\log^3 n \log k)$ w.h.p. The span is dominated by deleting up to k edges from A_i for each i (Line 7) for a cost of $O(\log^3 n \log k)$ span w.h.p. according to Theorem 4.5.

Space: The global tree F and each local RC tree T_i takes $O(n)$ space w.h.p., and each pair of local buffers $(B_{D,i}, B_{I,i})$ takes $O(n)$ space since they store the symmetric difference between the trees F and T_i. Each A_i has $O(\min\{2^i, n\})$ vertices and $O(2^i)$ edges, and the initialization strategy for the local graphs in Update leaves A_i empty for $i > \log m$. Applying Theorem 4.5 on each A_i gives a total space usage of $\sum_{i=1}^{\log m} O(2^i + \min\{2^i, n\} \log^3 n) = O(m + \min\{m, n \log n\} \log^3 n)$. □

5.3 Example

Table 2 displays an example of how the local graphs change as the global graph G changes. Though the algorithm does not actually store the non-tree edges of G or the uncompressed forms of the local graphs A_0 and A_1, they are displayed in the table for clarity.

In the first row of the table, we initialize a graph with four vertices u, v, x, and y.

In the second row of the table, we call BatchInsert on the five edges $(\{u, v\}, 4)$, $(\{u, x\}, 2)$, $(\{v, y\}, 3)$, $(\{u, y\}, 5)$, and $(\{v, x\}, 6)$. BatchInsert then invokes Update on the two non-tree edges $\{u, y\}$ and $\{v, x\}$. Placing these edges in A_0 would violate the edge-count invariant that A_0 has at most $2^0 = 1$ local non-tree edges, but we can place them in A_1 because it can have $2^1 = 2$ non-tree edges. The Update call hence initializes A_1 on the non-tree edges along with the current global tree. In this case, the compressed form of A_1 is the same as the uncompressed form—every vertex has an incident level-1 non-tree edge and cannot be compressed out.

In the third row of the table, we call BatchInsert on the edge $(\{x, y\}, 1)$. The edge $\{u, v\}$ becomes a non-tree edge, and Update is invoked on it. This time, the edge can be placed in A_0 without violating the edge-count invariant. The Update call initializes A_0 on the non-tree edge along with the current global tree. The compressed form of A_0 keeps the vertices u and v since they are non-tree edge endpoints and splices out vertices x and y, replacing the path u–x–y–v with a compressed edge $\{u, v\}$ that has the same weight as the heaviest weight in the path.

In the fourth row of the table, we call BatchDelete on edges $\{u, x\}$ and $\{v, y\}$. We first delete them from local graphs A_0 and A_1. Local graph A_0 returns local replacement edge $\{u, v\}$, and A_1 returns local replacement edges $\{u, y\}$ and $\{v, x\}$. (Deleting $\{u, x\}$ and $\{v, y\}$, or even just deleting either of these edges individually, in the compressed form of A_0 means deleting the entire compressed edge representing the path u–x–y–v. This triggers the correct behavior of searching for a local replacement edge that reconnects u's connected component and v's connected component.)

Finally, BatchDelete invokes BatchInsert to re-insert all of the local replacement edges into the global graph. Edges $\{u, v\}$ and $\{u, y\}$ are global replacement edges that are inserted as tree edges, whereas edge $\{v, x\}$ remains a global non-tree edge. We invoke Update on $\{v, x\}$, which re-initializes A_0 with the non-tree edge.

6 Dynamic graph HAC lower bounds

In this section, we show lower bounds on dynamic graph HAC under edge insertions and deletions. Queries take two vertices s and t and a similarity threshold θ, and answer whether s and t are in the same cluster if we run agglomerative clustering until all cluster similarities are strictly below θ. Such queries provide limited information, but we will show that answering such queries is still difficult for complete linkage, weighted average linkage, and average linkage. We also consider a different type of query that asks how many clusters there are if we run HAC until a threshold θ—we refer to this problem as #HAC. All of our lower bounds hold for the special case where s, t, and θ are fixed across all queries.

Our bounds hold even for an approximate form of graph HAC. We use the approximation notion given by Lattanzi et al. [35]. In λ-approximate graph HAC with $\lambda \geq 1$, at an agglomeration step where the maximum similarity is $\mathcal{W}_{\max}$, the clustering process may merge any clusters with similarity at least $\mathcal{W}_{\max}/\lambda$. We assume $\lambda \leq \text{poly}(n)$ so that $\text{poly}(\lambda)$ fits in a constant number of words.

6.1 Background: other dynamic lower bounds

Abboud and Vassilevska Williams as well as Henzinger et al. showed lower bounds on several dynamic problems conditional on well-known conjectures [1, 27]. The conjectures include the strong exponential time hypothesis (SETH) [11], triangle detection requiring greater than linear work, 3SUM requiring quadratic work [22, 41], and online Boolean matrix-vector multiplication (OMv) requiring cubic work [27].

The studied dynamic problems include Chan's subset union problem (SubUnion) [12], subgraph connectivity (SubConn), and connected subgraph (ConnSub). In SubUnion, given a collection of sets $X = \{X_1, \ldots, X_t\}$ and $U := \bigcup_i X_i$, we maintain a subcollection $S \subseteq X$ under insertions and deletions to S while answering whether $\bigcup_{X_i \in S} X_i = U$. In SubConn, given an undirected graph, we maintain a subset of vertices S under insertions and deletions to S with queries answering whether query vertices s and t are connected in the subgraph induced by S [21]. ConnSub is SubConn with the query instead being whether the subgraph is connected.

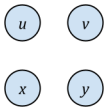
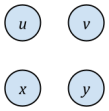
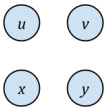
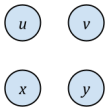
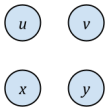
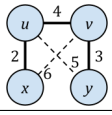
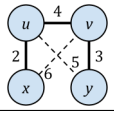
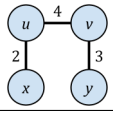
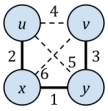
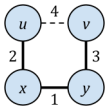
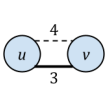
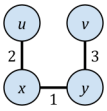
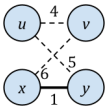
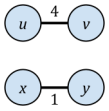
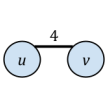
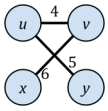
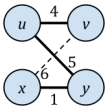
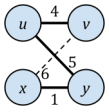
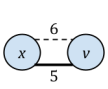
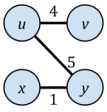
Operation	G	A_0 uncompressed	A_0 compressed	T_0	$B_{D,0}$ $B_{I,0}$	A_1 uncompressed	T_1	$B_{D,1}$ $B_{I,1}$
Initialize			(empty)		{} {}			{} {}
1st insert					{} {{u, v}, {u, x}, {v, y}}			{} {}
2nd insert					{} {}			{{u, v}} {{x, y}}
Delete (before insertion subroutine call)					{{u, x}, {v, y}} {}			{{u, v}, {u, x}, {v, y}} {{x, y}}
Delete (after insertion subroutine call)					{} {}			{{u, v}, {u, x}, {v, y}} {{u, v}, {u, y}{x, y}}

Table 2: This table walks through an example of the batch-dynamic MSF algorithm. The global graph G has four vertices. We first insert five edges into it, then insert one edge, and finally delete two edges. The bolded edges in G form the MSF (the global tree) F. The local graphs are A_0 and A_1, and the bolded edges within are the local tree edges. For $i \in \{0, 1\}$, the tree T_i is the RC tree for A_i. The buffers $B_{D,i}$ and $B_{I,1}$ represent the difference between T_i and F, i.e., $(T_i \setminus B_{D,i}) \cup B_{I,i} = F$. For brevity, we omit listing the weights of edges in the buffers. The compressed form of A_1 is the same as its uncompressed form throughout this example (except at initialization when the compressed form of A_1 is completely empty), and so we omit illustrating it. If a cell in the table is blank, that means it is the same as the cell in the row above.

Problem	Work lower bounds			Conjecture
	Preprocess	Update	Query	
SubUnion with $\lvert X\rvert = O(\log\lvert U\rvert)$	poly$(\lvert U\rvert)$	$\lvert U\rvert^{1-\varepsilon}$	$\lvert U\rvert^{1-\varepsilon}$	SETH
st-SubConn	$m^{1+\delta-\varepsilon}$	$m^{\delta-\varepsilon}$	$m^{2\delta-\varepsilon}$	Triangle
	$m^{4/3}$	$m^{\alpha-\varepsilon}$	$m^{2/3-\alpha-\varepsilon}$	3SUM
	poly(n)	$m^{1/2-\varepsilon}$	$m^{1-\varepsilon}$	OMv
ConnSub	poly(n)	$n^{1-\varepsilon}$	$n^{1-\varepsilon}$	SETH

Table 3: The table states conditional asymptotic work lower bounds for some dynamic problems [1, 27]. The values of ε, δ, and α follow the definitions in Table 1. The bounds hold for partially dynamic algorithms as well. The bounds are amortized in the fully dynamic case and are worst-case in the partially dynamic case.

Table 3 lists the lower bounds for these problems. The SubUnion bounds hold for the special case where $|X| = O(\log|U|)$, and the SubConn bounds hold for the special case of st-SubConn where s and t are fixed across all queries.

6.2 Statement of HAC lower bounds

The following theorem reduces SubConn to HAC and ConnSub to #HAC. It implies that the existing conditional lower bounds for st-SubConn apply directly to HAC under complete linkage or weighted average linkage. Similarly, the SETH gives the same lower bounds to #HAC as it does to ConnSub. We note that existing constructions reducing from triangle detection [1] and OMv [27] to SubConn also work when reducing to ConnSub, and so the conditional lower bounds for HAC based on triangle detection and OMv hardness apply to #HAC too.

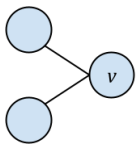
Theorem 6.1. *Suppose for some constant $c > 0$ that we can solve dynamic / incremental / decremental $O(n^c)$-approximate HAC under complete linkage or weighted average linkage in $p(m, n)$ preprocessing work, $u(m, n)$ update work, and $q(m, n)$ query work. Then we can solve dynamic / decremental / incremental SubConn with $\tilde{O}(m) + p(\tilde{O}(m), \tilde{O}(n))$ processing work, $u(\tilde{O}(m), \tilde{O}(n))$ update work, and $q(\tilde{O}(m), \tilde{O}(n))$ query work. The same relationship is also true between #HAC and ConnSub.*

In the full version of our paper [49], we also give a reduction from SubConn to average-linkage HAC. Due to the large size of the HAC instance resulting from the reduction, however, the only lower bound the reduction gives is $\Omega(m^{1/6-o(1)})$ update work and $\Omega(m^{1/3-o(1)})$ query work conditional on OMv hardness. Meanwhile, #HAC has $\Omega(m^{1/2-o(1)})$ update work and $\Omega(m^{1-o(1)})$ query work conditional on OMv hardness and also has lower bounds conditional on triangle detection hardness.

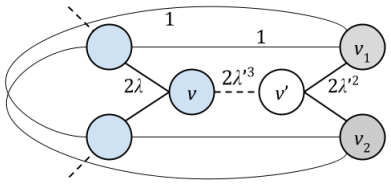
Finally, the following theorem states $\Omega(n^{1-o(1)})$ dynamic HAC lower bounds conditional on SETH. The lower bounds for complete-linkage and weighted-average-linkage #HAC come from Theorem 6.1. The remaining bounds come from reducing SubUnion to HAC and applying existing lower bounds on SubUnion.

Theorem 6.2. *Suppose that for some $\varepsilon > 0$ we can solve one of the following problems with* poly(n) *preprocessing work:*

- *for some constant c, fully dynamic $O(n^c)$-approximate HAC or $O(n^c)$-approximate #HAC under complete linkage or weighted average linkage with $O(n^{1-\varepsilon})$ amortized update and query work,*
- *the above problem in an incremental or decremental setting with $O(n^{1-\varepsilon})$ worst-case update and query work,*
- *for some $c \in [0, 1/5)$, fully dynamic $O(n^c)$-approximate average-linkage HAC with $O(n^{1-5c-\varepsilon})$ amortized update and query work,*
- *for some $c \in [0, 1/8)$, incremental or decremental $O(n^c)$-approximate average-linkage HAC with $O(n^{(1-8c)/2-\varepsilon})$ worst-case update and query work,*



(a) A vertex v with two neighbors in an SubConn instance.



(b) The corresponding HAC instance adds a star graph with center v' and several leaves.

Figure 2: The figure displays the extra vertices added for a particular vertex v in Lemma 6.3's reduction from SubConn to weighted-average-linkage HAC.

- *for some $c \in [0, 1)$, fully dynamic $O(n^c)$-approximate average-linkage #HAC with $O(n^{1-c-\varepsilon})$ amortized update and query work,*
- *for some $c \in [0, 1/2)$, incremental or decremental $O(n^c)$-approximate average-linkage #HAC with $O(n^{1-2c-\varepsilon})$ worst-case update and query work.*

Then the SETH is false.

6.3 Proof of bounds

As examples, we will show two reductions that generate some of our lower bounds. We defer the remaining reductions to the full version of our paper [49].

Our first example reduces SubConn to weighted-average-linkage HAC.

Lemma 6.3. *[Part of Theorem 6.1] Let $\lambda \in [1, \text{poly}(n)]$. Suppose we can solve dynamic / incremental / decremental λ-approximate weighted-average-linkage HAC in $p(m, n)$ preprocessing work, $u(m, n)$ update work, and $q(m, n)$ query work. Then, letting $m' = m(1+\log\lambda)$ and $n' = n(1 + \log\lambda)$, we can solve dynamic / decremental / incremental SubConn with $O(m') + p(O(m'), O(n'))$ processing work, $u(O(m'), O(n'))$ update work, and $q(O(m'), O(n'))$ query work. The same relationship is also true between #HAC and ConnSub.*

Proof. Suppose we are given an unweighted graph $G = (V, E)$ and we want to solve SubConn or ConnSub, maintaining some dynamic subset of vertices S. Set $\theta = 2\lambda$. Define $\ell = \lceil \log(2\lambda) \rceil = O(1 + \log\lambda)$ and $\lambda' = \lambda + 1$.

Preprocessing: Construct a new, weighted graph G' by copying G and giving every edge a weight of 2λ. For every $v \in V$, add a star graph to G' consisting of a center vertex v' and ℓ leaves $v_1, \ldots, v_\ell$ with weight-$2\lambda'^2$ edges. For each v_i, create weight-1 edges to each vertex in $N(v)$. Connect v to v' with weight $2\lambda'^3$ if $v \notin S$. Figure 2 illustrates this construction. The graph G' has $O(n(1 + \log(\lambda)))$ vertices and $O(m(1 + \log(\lambda)))$ edges. Initialize dynamic HAC on G' with $\theta = 2\lambda$.

Update: Simulate adding or removing a vertex v in S by removing or adding the weight-$2\lambda'^3$ edge $\{v, v'\}$.

Query: If we are reducing SubConn to HAC, return whether s and t are in the same cluster given similarity threshold θ. If we are reducing ConnSub to #HAC, then return whether the number of clusters is $|V \setminus S| + 1$.

Correctness: Consider running HAC until similarity threshold $\theta = 2\lambda$. Due to the preprocessing and the update strategy, every $v \in V \setminus S$ has a weight-$2\lambda'^3$ edge to v'. These edges merge first, and then all of the weight-$2\lambda'^2$ edges merge. This puts each $v \in V \setminus S$ in a cluster $\{v, v', v_1, v_2, \ldots, v_\ell\}$, where the incident edges connect to $N(v)$ with weight $1 + (2\lambda - 1)2^{-\ell} < 2$. To see why the weight is $1 + (2\lambda - 1)2^{-\ell}$, consider $v \in V \setminus S$ and suppose without loss of generality that the weight-$2\lambda'^2$ edges for v' merge in the order $v_1, v_2, \ldots, v_\ell$. Inductively, for $i = 0, 1, \ldots, \ell$, the incident edges on v''s cluster have weight $1 + (2\lambda - 1)2^{-i}$ after v_i merges with v''s cluster, where $v_0 = v$. The base case is prior to merging, where the weight is $1 + (2\lambda - 1)2^0 = 2\lambda$, which is correct by construction.

Therefore, the clusters for each $v \in V \setminus S$ do not participate in any more merges when clustering until threshold θ. The remaining vertices cluster into their connected components in the subgraph induced by S, and a HAC query answers SubConn correctly. A #HAC query will give the number of connected components in the subgraph plus $|V \setminus S|$ (one cluster for each $v \in V \setminus S$). □

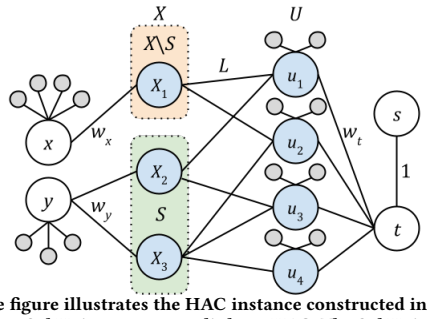


Figure 3: The figure illustrates the HAC instance constructed in Lemma 6.4's reduction from SubUnion to average-linkage HAC. The SubUnion instance in this example has $X = \{X_1 = \{u_1, u_2\}, X_2 = \{u_1, u_3\}, X_3 = \{u_2, u_3, u_4\}\}$ and currently has $S = \{X_2, X_3\}$. We reduce the number of stars' leaves (gray vertices) displayed for cleanliness (e.g., x should actually have dozens of leaves).

Our second example reduces SubUnion to average linkage HAC.

Lemma 6.4 (Part of Theorem 6.2)**.** *Let $\lambda \in [1, \text{poly}(n)]$. Suppose we can solve dynamic λ-approximate average-linkage HAC in $p(m, n)$ preprocessing work, $u(m, n)$ update work, and $q(m, n)$ query work. Then we can solve dynamic SubUnion with with $p(m', n')$ processing work, $u(m', n')$ update work, and $q(m', n')$ query work where m' and n' are $O(\lambda^5|U||X| + \lambda^2|X|^2)$.*

If we can solve incremental / decremental λ-approximate average-linkage HAC, then then the bounds hold for decremental / incremental SubUnion with m' and n' being $O(\lambda^8|U|^2|X| + \lambda^5|U||X|^2 + \lambda^2|X|^3)$.

Proof. Suppose we are given a SubUnion instance (X, U) with subset $S \subseteq X$. We focus on the case where we have a fully dynamic algorithm and defer the partially dynamic case to the full version of our paper [49]. Define $\theta = 1$ as well as the following constants:

$$w_t = (\lambda + 1)\lambda = O(\lambda^2),$$
$$\ell_y = \lambda w_t |U| = O(\lambda^3|U|),$$
$$L = (\lambda + 1)^2\lambda(\ell_y + 1 + |X| + \lambda|U|) = O(\lambda^6|U| + \lambda^3|X|),$$
$$\ell_x = |X|L/\lambda = O(\lambda^5|U||X| + \lambda^2|X|^2),$$
$$w_y = (\ell_y + |X|)L + 1 = O(\lambda^9|U|^2 + \lambda^6|U||X| + \lambda^3|X|^2),$$
$$w_x = \lambda(\ell_x + |X|)L + 1 = O(\lambda^{12}|U|^2|X| + \lambda^9|U||X|^2 + \lambda^6|X|^3).$$

Preprocessing: Figure 3 illustrates the graph that we will construct. Create a graph G with a vertex representing each $X_i \in X$, a vertex representing each $u \in U$, and a weight-L edge $\{X_i, u\}$ for each $u \in X_i$ for each $X_i \in X$. Make each $u \in U$ a center of a star graph with $\lambda - 1$ leaves connected with weight w_x. Add a star graph with center y and ℓ_y leaves connected to the center with weight w_y. Add a weight-w_y edge from y to each $X_i \in S$. Add another star graph

with center x and ℓ_x leaves connected to the center with weight w_x. Add a weight-w_x edge from x to each each $X_i \in X \setminus S$. Add two more vertices s and t with a weight-1 edge $\{s, t\}$, and add a weight-w_t edge $\{t, u\}$ for each $u \in U$. This graph has $O(\ell_x)$ vertices and edges. Initialize HAC on this graph.

Update: Simulate adding X_i to S by adding a weight-w_y edge from X_i to y and removing the weight-w_x from X_i to x. Similarly, simulate removing X_i from S by removing edge $\{X_i, y\}$ and adding edge $\{X_i, x\}$.

Query: Query whether s and t are in the same cluster when performing HAC until to threshold $\theta = 1$. If yes, then return that S covers U; otherwise, return that S does not cover U.

Correctness: Consider running HAC until threshold θ. The weights w_x and w_y are so large that all weight-w_x and weight-w_y edges merge before any weight-L edges. Let C_x denote x's cluster (containing $X \setminus S$) and C_y denote y's cluster (containing S). From this point onwards, the similarity between C_x to another cluster C is bounded above by considering the worst case where every X_i in C_x has a weight-L edge to every $u \in U$ contained in C:

$$\frac{|X \setminus S||C \cap U|L}{|C_x||C|} \leq \frac{|X \setminus S|L}{|C_x|\lambda} < \frac{|X|L}{\ell_x \lambda} = 1,$$

where the first inequality used the fact that $\lambda|C \cap U| \leq |C|$ due to each $u \in U$ being in a star of size λ. Hence, C_x does not participate in any more merges until near the end of the agglomeration process, at which point those merges will not affect correctness. On the other hand, consider C_y, which has weight-L edges connecting it to every $u \in U$ covered by S. Each $u \in U$, until it merges with C_y, is in a size-λ cluster consisting only of the star centered on u. The similarity between C_y and some $u \in U$ covered by S and not yet merged with C_y is always at least

$$\frac{L}{(\ell_y + 1 + |X| + \lambda|U|)\lambda} = (\lambda + 1)^2. \tag{1}$$

In comparison, the similarity between t and another adjacent cluster C is 1 if $C = \{s\}$ and is otherwise at most

$$\frac{w_t|C \cap U|}{|C|} \leq \frac{w_t}{\lambda} = \lambda + 1 \tag{2}$$

where the first inequality again uses the inequality $\lambda|C \cap U| \leq |C|$. Comparing Equation (1) to Equation (2) shows that C_y merges with all $u \in U$ covered by S before t merges with anything.

Now consider what t merges with. In the case where S does not cover all of U, inequality (2) is tight for every cluster C representing an uncovered $u \in U$. In particular, the similarity between t and such a cluster is λ times greater than the similarity between t and s or between C_x and any cluster. Therefore, t merges with some uncovered u's star rather than merging with s and is in a cluster of size $\lambda + 1$. The similarity between t's cluster and s falls to $1/(\lambda + 1)$, and t and s never merge. Hence a query returns the correct result in this case.

In the case where S covers all of U, the only adjacent clusters to t are $\{s\}$ and C_y. The similarity between t and C_y is

$$\frac{w_t|U|}{\ell_y + 1 + |S| + \lambda|U|} < \frac{w_t|U|}{\ell_y} = 1/\lambda,$$

which is λ times less than the similarity between t and s. Hence t and s merge, and a query returns the correct result in this case too. □

To turn Lemma 6.4 into the lower bounds in Theorem 6.2, we need the following lemma.

Lemma 6.5. *Let a and b be constants. Let $\mathcal{P}$ be some dynamic problem. Suppose that given a dynamic / incremental / decremental SubUnion instance with $|X| = O(\log|U|)$, for any value of $\lambda \geq 1$, we can solve the instance by efficiently converting it to an instance of λ-approximate $\mathcal{P}$ of size $n' = \tilde{O}(\lambda^a|U|^b)$. Assuming the SETH holds, for any $c < 1/a$, the update or query work of an $O(n^c)$-approximate $\mathcal{P}$ algorithm with $\mathrm{poly}(n)$ preprocessing work is $\Omega(n^{(1-ac)/b-o(1)})$ amortized / worst-case / worst-case.*

Proof. Set $\lambda = \tilde{\Theta}(|U|^{bc/(1-ac)})$ so that $n' = \tilde{O}(\lambda^a|U|^b) = \tilde{O}(|U|^{abc/(1-ac)}|U|^b) = \tilde{O}(|U|^{abc/(1-ac)}|U|^{(b-abc)/(1-ac)}) = \tilde{O}(|U|^{b/(1-ac)})$ and $\lambda \geq n'^c$. Solve the SubUnion instance by generating a λ-approximate instance of $\mathcal{P}$ of size n' and running an $O(n'^c)$-approximate algorithm for $\mathcal{P}$. The update and query time for the algorithm cannot both be $O(n'^{(1-ac)/b-\Omega(1)}) = \tilde{O}(|U|^{1-\Omega(1)})$ because such a work bound for SubUnion is impossible if the SETH holds. □

For example, Lemma 6.4 shows that SubUnion with $|X| = O(\log|U|)$ can be solved by running dynamic λ-approximate average-linkage HAC on a graph with $\tilde{O}(\lambda^5|U|)$ vertices and edges. Setting $(a, b) = (5, 1)$ in the lemma above gives the conditional lower bound on fully dynamic average-linkage HAC stated in Theorem 6.2.

7 Conclusion

In this paper, we gave a fully dynamic MSF algorithm that processes a batch of k updates in $O(k \log^6 n)$ expected amortized work and $O(\log^3 n \log k)$ span w.h.p. This gives a batch-dynamic algorithm that can answer queries about single-linkage graph HAC clusters. We also showed that graph HAC requires polynomial query or update time for other common linkage functions unless we can break long-standing computational complexity conjectures. This suggests that future work on dynamic HAC algorithms for these linkage functions may wish to avoid targeting worst-case inputs.

For future work, it would be desirable to reduce the running time of the MSF algorithm further. Can we match the $O(k \log^4 n)$ work of the sequential HDT MSF algorithm? Can we match the $O(\log^3 n)$ span of of Acar et al.'s best dynamic connectivity bounds? It would also be interesting to design practical implementations of our MSF algorithm. Finally, it would be interesting to find restricted input classes on which we can break the lower bounds shown in this paper.

Acknowledgments

We thank the reviewers for their helpful feedback. This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, NSF Award #CCF-2103483, Google Faculty Research Award, Google Research Scholar Award, FinTech@CSAIL Initiative, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*. 434–443.

[2] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. 2019. Parallel Batch-Dynamic Graph Connectivity. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 381–392.

[3] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel Batch-Dynamic Trees via Change Propagation. In *28th Annual European Symposium on Algorithms*, Vol. 173. 2:1–2:23.

[4] Umit A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. 2004. Dynamizing Static Algorithms, with Applications to Dynamic Trees and History Independence. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Vol. 531. 540.

[5] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 1997. Minimizing Diameters of Dynamic Trees. In *International Colloquium on Automata, Languages, and Programming*. 270–280.

[6] Daniel Anderson, Guy E. Blelloch, and Kanat Tangwongsan. 2020. Work-efficient Batch-Incremental Minimum Spanning Trees with Applications to the Sliding Window Model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 51–61.

[7] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. 2017. Affinity Clustering: Hierarchical Clustering at Scale. In *Advances in Neural Information Processing Systems*. 6864–6874.

[8] J.-P. Benzécri. 1982. Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données* 7, 2 (1982), 209–218.

[9] Guy E Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[10] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. 253–264.

[11] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. 2009. The Complexity of Satisfiability of Small Depth Circuits. In *International Workshop on Parameterized and Exact Computation*. Springer, 75–85.

[12] Timothy M Chan. 2006. Dynamic subgraph connectivity with geometric applications. *SIAM J. Comput.* 36, 3 (2006), 681–694.

[13] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. 2020. A Deterministic Algorithm for Balanced Cut with Applications to Dynamic Connectivity, Flows, and Beyond. In *Proceedings of the 61st Annual Symposium on Foundations of Computer Science*. 1158–1167.

[14] Vincent Cohen-Addad, Varun Kanade, Frederik Mallmann-trenn, and Claire Mathieu. 2019. Hierarchical Clustering: Objective Functions and Algorithms. *J. ACM* 66, 4 (2019).

[15] Sajal K Das and Paolo Ferragina. 1995. Parallel Dynamic Algorithms for Minimum Spanning Trees. (1995).

[16] Sanjoy Dasgupta. 2016. A Cost Function for Similarity-Based Hierarchical Clustering. In *ACM Symposium on Theory of Computing (STOC)*. 118–127.

[17] Erik Demaine. 2004. Lecture Notes on Skip Lists. https://courses.csail.mit.edu/6.046/spring04/handouts/skiplists.pdf. [Online; accessed 08-January-2022].

[18] Laxman Dhulipala, David Eisenstat, Jakub Łącki, Vahab Mirrokni, and Jessica Shi. 2021. Hierarchical Agglomerative Graph Clustering in Nearly-Linear Time. In *International Conference on Machine Learning*. 2676–2686.

[19] Paolo Ferragina and Fabrizio Luccio. 1994. Batch dynamic algorithms for two graph problems. In *International Conference on Parallel Architectures and Languages Europe*. 713–724.

[20] Pasi Franti, Olli Virmajoki, and Ville Hautamaki. 2006. Fast Agglomerative Clustering Using a k-nearest Neighbor Graph. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 11 (2006), 1875–1881.

[21] Daniele Frigioni and Giuseppe F Italiano. 2000. Dynamically Switching Vertices in Planar Graphs. *Algorithmica* 28, 1 (2000), 76–103.

[22] Anka Gajentaan and Mark H Overmars. 1995. On a class of $O(n^2)$ problems in computational geometry. *Computational geometry* 5, 3 (1995), 165–185.

[23] Joseph Gil, Yossi Matias, and Uzi Vishkin. 1991. Towards a Theory of Nearly Constant Time Parallel Algorithms. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*. 698–710.

[24] John C Gower and Gavin JS Ross. 1969. Minimum Spanning Trees and Single Linkage Cluster Analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 18, 1 (1969), 54–64.

[25] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. 24–34.

[26] Timothy C Havens, James C Bezdek, James M Keller, Mihail Popescu, and Jacalyn M Huband. 2009. Is VAT really single linkage in disguise? *Annals of Mathematics and Artificial Intelligence* 55, 3 (2009), 237–251.

[27] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 21–30.

[28] Monika Rauch Henzinger and Valerie King. 1995. Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*. 519–527.

[29] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *J. ACM* 48, 4 (2001), 723–760.

[30] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. 2015. Faster fully-dynamic minimum spanning forest. In *23rd Annual European Symposium on Algorithms*. 742–753.

[31] Joseph JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison-Wesley.

[32] George Karypis, Eui-Hong Han, and Vipin Kumar. 1999. Chameleon: Hierarchical Clustering using Dynamic Modeling. *Computer* 32, 8 (1999), 68–75.

[33] Ari Kobren, Nicholas Monath, Akshay Krishnamurthy, and Andrew McCallum. 2017. A Hierarchical Algorithm for Extreme Clustering. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 255–264.

[34] Tsvi Kopelowitz, Ely Porat, and Yair Rosenmutter. 2018. Improved Worst-Case Deterministic Parallel Dynamic Minimum Spanning Forest. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 333–341.

[35] Silvio Lattanzi, Thomas Lavastida, Kefu Lu, and Benjamin Moseley. 2019. A Framework for Parallelizing Hierarchical Clustering Methods. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. 73–89.

[36] Aditya Krishna Menon, Anand Rajagopalan, Baris Sumengen, Gui Citovsky, Qin Cao, and Sanjiv Kumar. 2019. Online hierarchical clustering approximations. *arXiv preprint arXiv:1909.09667* (2019).

[37] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. 1994. Complexity models for incremental computation. *Theoretical Computer Science* 130, 1 (1994), 203–236.

[38] Nicholas Monath, Ari Kobren, Akshay Krishnamurthy, Michael R Glass, and Andrew McCallum. 2019. Scalable Hierarchical Clustering with Tree Grafting. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1438–1448.

[39] Benjamin Moseley and Joshua R. Wang. 2017. Approximation Bounds for Hierarchical Clustering: Average Linkage, Bisecting K-means, and Local Search. In *Advances in Neural Information Processing Systems*. 3094–3103.

[40] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. 2017. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science*. 950–961.

[41] Mihai Patrascu. 2010. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*. 603–610.

[42] Shaunak Pawagi and Owen Kaser. 1993. Optimal parallel algorithms for multiple updates of minimum spanning trees. *Algorithmica* 9, 4 (1993), 357–381.

[43] Seth Pettie and Vijaya Ramachandran. 2002. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.* 31, 6 (2002), 1879–1895.

[44] Xiaojun Shen and Weifa Liang. 1993. A parallel algorithm for multiple edge updates of minimum spanning trees. In *Proceedings of the Seventh International Parallel Processing Symposium*. 310–317.

[45] Daniel D Sleator and Robert Endre Tarjan. 1983. A data structure for dynamic trees. *Journal of computer and system sciences* 26, 3 (1983), 362–391.

[46] Yihan Sun. 2019. *Join-based Parallel Balanced Binary Trees*. Ph.D. Dissertation. Carngie Mellon University.

[47] Yihan Sun, Daniel Ferizovic, and Guy E Belloch. 2018. PAM: Parallel Augmented Maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 290–304.

[48] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. 2019. Batch-Parallel Euler Tour Trees. In *Proceedings of the Meeting on Algorithm Engineering and Experiments*. 92–106.

[49] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Minimum Spanning Forest and the Efficiency of Dynamic Agglomerative Graph Clustering. *arXiv preprint arXiv:2205.04956* (2022).

[50] Qi Zhang and Wei Wang. 2007. An efficient algorithm for approximate biased quantile computation in data streams. In *Proceedings of the 16th ACM International Conference on Information and Knowledge Management*. 1023–1026.