Supplementary Materials: Hierarchical Agglomerative Graph Clustering in Nearly-Linear Time

A Overview of Materials Included

1. In Section B, we restate key definitions and algorithms from our paper.
2. Then, in Section C, we present details about some of our data structures such as mergeable heaps, which are used in our proofs.
3. Next, in Section D, we present deferred proofs regarding the efficiency of our algorithms.
4. Lastly, in Section E, we present deferred experimental details from our evaluation.

B Algorithms and Definitions

For ease of reference, we restate a few keys definitions and algorithms that are used in the rest of the appendix.

B.1 Linkage Measures

Let the *best-neighbor* of a cluster X be $\text{argmax}_{Y \in N(X)} \mathcal{W}(X, Y)$. We call the edge connecting X and its best-neighbor Y the *best-edge* of X. We say that a linkage measure is *reducible* [Ben82], if for any three clusters X, Y, Z where X and Y are mutual best-neighbors, it holds that $\mathcal{W}(X \cup Y, Z) \geq \max(\mathcal{W}(X, Z), \mathcal{W}(Y, Z))$. Our framework yields exact HAC algorithms for any *reducible* linkage measure that also satisfies the following property.

Definition B.1. *A linkage measure is called* triangle-based *if it satisfies the following property. Consider any step of the algorithm which merges clusters B and C into a cluster $B \cup C$. Let A be a cluster distinct from B and C. Then, if edge (A, C) does not exist, $\mathcal{W}(A, B) = \mathcal{W}(A, B \cup C)$.*

B.2 Algorithms

Most of our algorithms use the Merge routine to merge two clusters, which is given below (Algorithm 1). Next, we provide the two main HAC implementations in our framework, which are based on the classic nearest-neighbor chain and heap-based methods (Algorithm 2 and Algorithm 3 respectively).

C Neighbor-Heap Details

Augmented Heaps. A simple implementation of neighbor-heaps is to store the edges incident to a cluster using an augmented binary tree, or an augmented heap. Given a binary tree storing key-value entries, and a function f taking an entry and yielding a real-valued priority, we can obtain a max-heap (min-heap) by setting the initial augmented value of each vertex to its priority calculated

Algorithm 1 MERGE$(A, B, \mathcal{L})$

Input: Active clusters A and B, triangle-based linkage $\mathcal{L}$.
Output: Cluster id of the remaining active cluster.

1: $(A, B) = (\text{argmin}(d(A), d(B)), \text{argmax}(d(A), d(B))$.
2: Remove B from HEAP(A) (and vice versa).
3: HEAP(B) = HEAP$(A) \cup$ HEAP(B) (using $\mathcal{L}$ to merge the weights of $C \in$ HEAP$(A) \cap$ HEAP(B))
4: Mark cluster A as inactive.
5: For each $C \in$ HEAP(A), update the cluster-id from A to B in HEAP(C), using $\mathcal{L}$ to merge if $B \in$ HEAP(C).
6: Return B.

Algorithm 2 GraphHAC-Chain$(G = (V, E, w), \mathcal{L})$

Input: Edge weighted graph, G, triangle-based linkage $\mathcal{L}$.
Output: Dendrogram D for $\mathcal{L}$-HAC.

```
1: for each cluster v ∈ V do
2:   if v is active then
3:     Initialize stack S, initially containing only v.
4:     while S is not empty do
5:       Let t = Top(S).
6:       Let (t, b, W(t, b)) = BestEdge(t).
7:       if b is already on S then
8:         Pop(S).
9:         Merge(t, Top(S)).  [Algorithm 1]
10:        Pop(S).
11:      else
12:        Push b onto S.
```

using f, and inductively setting the augmented values for internal vertices using max (min) of their augmented value, and the augmented values of their two children. We refer to [SFB18] for details on implementing augmented binary trees.

The BESTEDGE operation can be implemented by having the augmented value at each tree vertex (corresponding to an edge of the graph) be the edge weight, and the augmentation function to max. Extracting the best-edge can then be done using a find-like function which finds a (key, value) pair in the tree that exhibits the overall augmented value of the tree. Another way is to set the augmented value to a pair of the edge weight and the neighbor id, and have the augmentation function to take a lexicographic maximum. The best-edge in this approach is simply the augmented value at the root of the tree. In our implementations, we use the former approach, since it is more space-efficient than the latter approach. Both implementations cost $O(\log n)$ work per BESTEDGE operation.

The UNION operation can be implemented by using UNION on the underlying augmented binary trees. The algorithm from [BFS16] (which is implemented in the PAM library [SFB18]) merges two trees of size n, m with $n \leq m$ in $O(n \log(m/n + 1))$ time, which is asymptotically optimal for comparison based algorithms [BT79].

Algorithm 3 GraphHAC-Heap($G = (V, E, w), \mathcal{L}$)

Input: Edge weighted graph, G, triangle-based linkage $\mathcal{L}$.
Output: Dendrogram D for $\mathcal{L}$-HAC.

Let H be a max-heap storing the highest-weight edge incident to each active cluster in the graph.
while $|H| > 1$ **do**
  Let $e = (u, v, \mathcal{W}(u, v))$ be the max edge in H.
  Delete e from H.
  if v is inactive **then**
    Let $e' = (u, v', \mathcal{W}(u, v')) = \textsc{BestEdge}(u)$.
    Insert e' into H.
  else
    $x = \textsc{Merge}(u, v)$. [Algorithm 1]
    Let $e' = (x, y, \mathcal{W}(x, y)) = \textsc{BestEdge}(x)$.
    Insert e' into H.

D Deferred Proofs

Lemma D.1. *Single-linkage, complete-linkage, and WPGMA-linkage are all triangle-based linkages.*

Proof. It is well known that all of these linkage rules are reducible (e.g., see [MRS08]). To show that they also satisfy Definition B.1, observe that the weight of an edge (A, B) can change after merging clusters B and C only when A is also connected to C. For example, for single-linkage, it is easy to see that the edges that we take the max over when calculating $\mathcal{W}(A, B \cup C) = \max_{(x,y)\in E(A,B\cup C)} w(x, y)$ can affect the result only when $\mathcal{W}(A, C) > \mathcal{W}(A, B)$. Similar calculations show that complete-linkage and WPGMA-linkage are triangle-based linkages. □

D.1 Framework Analysis

Lemma D.2. *Let $m_1, \ldots, m_{n-1}$ be the sequence of merge operations performed by an algorithm where $m_i = (u_i, v_i)$. Let $\mathsf{Cost}(m_i) = \min(d(u_i), d(v_i))$, where $d(u_i)$ and $d(v_i)$ are the degrees of the clusters when they are merged. Then, the total cost $\sum_{i=1}^{n-1} \mathsf{Cost}(m_i) = O(m \log n)$.*

Proof. The proof is by a charging argument. Following the definition of Cost, assign a token to each edge in $N(\text{argmin}(d(u_i), d(v_i)))$ for each i. We do not undercount the cost, since we assign $\mathsf{Cost}(m_i)$ tokens at each step i to the edges. We now bound the total cost by bounding the number of tokens that can be assigned to an edge.

We conceptually add an extra *total-edges* variable, $\mathcal{T}(u)$ to the data structures storing the vertex neighborhoods, $N(u)$. This variable simply stores the total number of edges that have been merged into this tree. At the start of the algorithm, $\mathcal{T}(u) = d(u)$. When two clusters u and v merge, if v is the cluster remaining active after the merge, $\mathcal{T}(v)$ is incremented by $\mathcal{T}(u)$. Let A be the set of active clusters. It is easy to check that $\sum_{c\in A} \mathcal{T}(c) = 2m$ at all points in the algorithm.

Next, we bound the maximum number of tokens assigned to an edge by observing that each time an edge has a token assigned to it in some step i, the total-edges of the set containing it doubles. Since $\mathcal{T}(u)$ of a set u can grow to at most $2m$, each edge can receive at most $O(\log m) = O(\log n)$ tokens, and thus the total cost is $O(m \log n)$. □

Theorem D.3. *There are deterministic implementations of the chain-based and heap-based algorithms that run in $O(m \log^2 n)$ time for any triangle-based linkage $\mathcal{L}$.*

Proof. We first bound the time-complexity for the merge steps that both algorithms have in common. Both algorithms perform $n-1$ merge operations, whose total cost is $O(m \log n)$ using Lemma D.2. To translate this cost measure back to time-complexity, we note that the total time taken for the i-th merge step using the deterministic neighbor-heap implementation is at most $O(\mathsf{Cost}(m_i) \log(n/\mathsf{Cost}(m_i) + 1)) = O(\mathsf{Cost}(m_i) \log n)$, and thus the total time to perform all merges is $O(m \log^2 n)$. We bound the time of the remaining algorithm-specific steps separately.

Chain-based Algorithm. We use a few well-known facts about this algorithm, namely that (i) each of the $2n-1$ clusters that appears in the dendrogram is pushed onto the stack exactly once, and (ii) the number of times BestEdge is called is $O(n)$. Therefore, the total time-complexity of these steps is $O(n \log n)$ and the overall time-complexity is $O(m \log^2 n)$.

Heap-based Algorithm. The remaining steps involve extracting edges from the global heap H. We analyze two types of edges that can be extracted: (i) edges whose remaining endpoint is inactive and (ii) edges whose remaining endpoint is active. There are at most $n-1$ type (ii) edges, since each type (ii) edge results in a merge, and so the time spent processing these edges is $O(n \log n)$. Next, for the type (i) edges, observe that each such edge can be charged to the deactivated endpoint, v, and that the total number of charges for a cluster v is at most its degree at the time it was deactivated. Thus, the total cost for these edges is $\sum_{i=1}^{n-1} \mathsf{Cost}(m_i) = O(m \log n)$ (by Lemma D.2) and the time-complexity for these steps is $O(m \log^2 n)$ since each inactive edge takes $O(\log n)$ time to extract the current best edge from its active endpoint, and to update H. Thus, the overall time-complexity is $O(m \log^2 n)$. □

D.2 A Faster Randomized Chain-Based Algorithm

In this section we present a randomized implementation of our chain-based algorithm which runs in $O(m \log n)$ time.

Overview. There are two challenges posed by Algorithm 1 that we must implement more efficiently in order to achieve an $O(m \log n)$ time HAC algorithm.

1. The *merge-cost* from Lemma D.2 is $O(m \log n)$. Thus, in order to achieve $O(m \log n)$ time we must perform each merge operation in (amortized) $O(1)$ time per merged element.
2. The overall algorithm also performs $O(m \log n)$ *neighbor-updates* in Line 5 of Algorithm 1, which remove the id of a merged vertex from an active neighbor's neighbor-heap and relabel it to the id of the new neighbor. Thus, we must either handle these updates lazily, or also handle them in (amortized) $O(1)$ time per operation.

Our approach to handle (1) is to use a faster randomized implementation of neighbor-heaps which we outline below. The high-level idea is to use an efficient meldable heap, such as a Fibonacci heap or Leftist heap in conjunction with hash tables. We deal with (2) by *eagerly* updating the hash tables of our neighbors when performing a merge, but *lazily* updating the IDs stored in the meldable heap, except when we identify an edge that is being merged. The overall cost of the hash table updates is $O(m \log n)$ time in expectation. Although the updates to the heaps cost $O(\log n)$ time each, since they require deleting two existing elements and reinserting a new merged element, each of these updates can be assigned uniquely to an edge in the original graph, and thus the overall time complexity for these updates is also $O(m \log n)$. We now provide a detailed description of our approach.

D.2.1 Neighbor-Heaps using Meldable Heaps and Hashing

We give an alternative implementation of neighbor-heaps, which is asymptotically faster than the augmented-heap based implementation at the cost of using randomization. The idea is to use a heap data structure that supports efficient *melding*, such as Fibonacci heaps [FT87], in combination with a hash table. The neighbor-heap representation for a cluster v is a pair of a heap and a hash-table where the neighbors of v are stored in both data structures. Let $\mathcal{H}(A)$ denote the heap and $\mathcal{Q}(A)$ denote the table for a cluster A. The elements in $\mathcal{H}(A)$ are pairs of a cluster-id, C, and the associated weight of this edge $\mathcal{W}(A, C)$. The priority of an element is just the weight. The elements in $\mathcal{Q}(A)$ are triples of a key (a cluster-id), C, the associated weight of this edge $\mathcal{W}(A, C)$, and a pointer to the location of the element for C in $\mathcal{H}(A)$.

The MELD operation on two mergeable heaps $\mathcal{H}_1, \mathcal{H}_2$ can be done in $O(1)$ time. Note that this operation *does not* detect elements in $\textsc{Keys}(\mathcal{H}_1) \cap \textsc{Keys}(\mathcal{H}_2)$, which is why we also store the elements in both heaps in a hash-table (which implements intersection efficiently).

We also define a T-MERGE operation on two hash-tables in neighbor-heaps, $\mathcal{Q}(A)$, $\mathcal{Q}(B)$, which works as follows. Without loss of generality let $|\mathcal{Q}(A)| \leq |\mathcal{Q}(B)|$. We map over the elements in $\mathcal{Q}(A)$, and insert them into the larger size table. If an key C appears in both tables, then we merge this edge using the linkage function $\mathcal{L}$. We also append C and the locations of C in both heaps to an array O that collects all of the locations for $C \in \mathcal{Q}(A) \cap \mathcal{Q}(B)$. The T-MERGE operation returns O and $\mathcal{Q}(B)$. T-MERGE on two tables $\mathcal{Q}(A)$, $\mathcal{Q}(B)$ runs in $O(\min(|\mathcal{Q}(A)|, |\mathcal{Q}(A)|))$ expected time.

D.2.2 Merging Clusters

Next, we present how two clusters are merged using the randomized neighbor-heap (Algorithm 4). Algorithm 4 is similar to our original Merge algorithm, Algorithm 1 with a few key differences.

Algorithm 4 $\textsc{FastMerge}(A, B, \mathcal{L})$

Input: Active clusters A and B, triangle-based linkage $\mathcal{L}$.
Output: Cluster id of the remaining active cluster.

1: $(A, B) = (\text{argmin}(d(A), d(B)), \text{argmax}(d(A), d(B)))$
2: Remove B from $\mathcal{H}(A)$ and $\mathcal{Q}(A)$ (and vice versa).
3: $(O, \mathcal{Q}(B)) = \text{T-}\textsc{Merge}(\mathcal{Q}(A), \mathcal{Q}(B))$ ▷ O holds heap-locations for $C \in \mathcal{Q}(A) \cap \mathcal{Q}(B)$
4: **for** each $(C, L_A, L_B) \in O$ **do**
5: Delete C from $\mathcal{H}(A)$ and $\mathcal{H}(B)$ using L_A and L_B to find these elements.
6: Insert C into $\mathcal{H}(B)$ with the weight merged using $\mathcal{L}$. Let the pointer to this element be L'.
7: Update the location of C in $\mathcal{Q}(B)$ to L'.
8: $\mathcal{H}(B) = \textsc{Meld}(\mathcal{H}(A), \mathcal{H}(B))$. ▷ Before the meld, $\textsc{Keys}(\mathcal{H}(A)) \cap \textsc{Keys}(\mathcal{H}(B)) = \emptyset$.
9: **for** $C \in \textsc{Keys}(\mathcal{Q}(A))$ **do**
10: Update cluster-id from A to B in $\mathcal{Q}(C)$ and $\mathcal{H}(C)$. If $B \in \mathcal{Q}(C)$, use $\mathcal{L}$ to merge the edge weights and update both $\mathcal{Q}(C)$ and $\mathcal{H}(C)$.
11: Mark cluster A as inactive.
12: Return B.

First, Line 2 removes the IDs of the merged clusters from both the heaps and hash-tables for each of the merging clusters. Next, on Line 3 the algorithm merges the hash-tables of both clusters using the T-MERGE routine described above. The result is a sequence O of triples containing the cluster-id, and two heap-locations of $C \in \textsc{Keys}(\mathcal{Q}(A)) \cap \textsc{Keys}(\mathcal{Q}(B))$, and the newly merged table, $\mathcal{Q}(B)$. The algorithm then loops over these clusters C with edges to both A and B, and the location of

these edges in $\mathcal{H}(A)$ and $\mathcal{H}(B)$ (Lines 4–7). For each such cluster, the algorithm first deletes C from $\mathcal{H}(A)$ and $\mathcal{H}(B)$ using the given locations (Line 5). It then inserts C into $\mathcal{H}(B)$ with the updated weight of this edge (Line 6). Note that at this point, $\textsc{Keys}(\mathcal{H}(A)) \cap \textsc{Keys}(\mathcal{H}(A)) = \emptyset$. After the loop, the algorithm first melds the two heaps (Line 8). The last step is to update the neighbor-heaps of neighbors of A (the deactivated cluster). The algorithm iterates over all neighbors C of A on Line 9, and on Line 10, updates the id from A to B in $\mathcal{Q}(C)$ and $\mathcal{H}(C)$. Note that the location of A in $\mathcal{H}(C)$ is stored $\mathcal{Q}(C)$. If C also has an edge to B, it sets the weight of the (C, B) edge to the updated weight in $\mathcal{Q}(C)$. It also deletes A and B from $\mathcal{H}(C)$ and reinserts B into $\mathcal{H}(C)$ with the correct weight. Finally, it marks A as inactive and returns the ID of the remaining active cluster, B.

D.2.3 Modifications to the Chain-Based Algorithm

Lastly, we discuss how to modify the chain-based algorithm to obtain an $O(m \log n)$ time HAC for triangle-based linkage. The algorithm is identical to Algorithm 2 with the only differences being the representation of the neighbor-heap data structures, and the merge routine. Specifically, the call to Merge on Line 9 uses the FastMerge algorithm (Algorithm 4). As Lemma D.4 shows, after a merge, the state of the neighbor-heap data structures corresponds to the state of the current graph induced by the active clusters, and thus we do not have to modify BestEdge.

Lemma D.4. *After a call to Algorithm 4, the adjacency information stored in the neighbor-heaps (both $\mathcal{H}(A)$ and $\mathcal{Q}(A)$) of all active clusters A is correct.*

Proof. The proof is by induction. Consider the k-th merge between two vertices A and B, and assume that the claim holds before this merge. Assume without loss of generality that A is deactivated and B remains active. The only clusters affected by the merge are $\{B\} \cup \{C \in N(A)\}$, since all neighbors in $N(B) \setminus N(A)$ have an edge to B with the same weight as before the merge.

First, we show that Algorithm 4 correctly updates the edges incident to B. The only edges that experience weight change are those in $N(A) \cap N(B)$, which the algorithm detects when performing T-Merge on Line 3. For each neighbor C in $N(A) \cap N(B)$, it deletes C from both $\mathcal{H}(A)$ and $\mathcal{H}(B)$ (Line 5) and reinserts C into $\mathcal{H}(B)$ with the correct weight. Finally, the location corresponding to C is updated in $\mathcal{Q}(B)$. Note that T-Merge also sets the weight of C correctly in $\mathcal{Q}(B)$. The remaining affected edges are new neighbors of B, which are correctly labeled and stored in $\mathcal{Q}(B)$ and $\mathcal{H}(B)$.

Second, we show that Algorithm 4 correctly updates the neighbor-heaps for $C \in N(A)$. It processes these neighbors in the for-loop on Line 9. If the neighbor C is not in $N(A) \cap N(B)$, it just updates the cluster-id from A to B in $\mathcal{H}(C)$, and leaves the location in $\mathcal{Q}(C)$ unchanged. Otherwise, for $C \in N(A) \cap N(B)$, it deletes A and B from $\mathcal{H}(C)$, updates the weight of the edge using $\mathcal{L}$ and reinserts B into $\mathcal{H}(C)$ (similarly for $\mathcal{Q}(C)$). Therefore, all of the neighbors $C \in N(A)$ reference B after Algorithm 4 finishes. □

Using Lemma D.4 we have that the state of the neighbor-heap data structures correspond to the current state of the graph induced by the active clusters after performing a merge operation. Combining the fact that the neighbor-heap data is always correct after a merge with the existing proof for the correctness of the chain-based algorithm suffices to show that our randomized implementation is correct. Next, we show that our approach is also efficient.

Theorem D.5. *There is a randomized implementation of the chain-based algorithm that runs in $O(m \log n)$ time in expectation for any triangle-based linkage $\mathcal{L}$.*

Proof. We follow the proof of Theorem D.3 and separately account for the cost of the merge steps, and the cost of the remaining steps in the algorithm.

The algorithm performs $n-1$ merge operations, with a total merge-cost of $O(m \log n)$ using Lemma D.2. To translate this cost measure to the time-complexity measure, we examine the two types of operations done inside of the FastMerge algorithm (Algorithm 4).

The first type of updates are those done on the hash-tables, $\mathcal{Q}(C)$ for a cluster C. Over all merges, there are $O(m \log n)$ such operations, which each cost $O(1)$ time in expectation, and thus the overall cost of the hash-table updates are $O(m \log n)$ in expectation.

The second type of updates are done on the heaps. First, the cost of melding the two heaps is $O(1)$ amortized using lazy Fibonacci heaps and $O(\log n)$ using Leftist or eager Fibonacci heaps. In either case, the overall cost of the meld operations is at most $O(n \log n)$. The remaining heap operations can be broken up further into two categories of heap updates which update the cluster-ids and weights of edges in the heaps.

1. Updates that only affect the cluster-ids of an edge cost $O(1)$ time each in expectation, since they are done by looking up the location of the edge in the hash-table, and updating the id of this element in $O(1)$ time.
2. Updates that change the weights of edges in the heap are more costly since they require deleting and reinserting elements from the heap, and deleting an element costs $O(\log n)$ time. However, updating the weight of an edge is done only when two clusters A, B merge and both A, B have an edge to a neighbor cluster C. We observe that we can charge the cost of this step to one of the original edges in the graph, and that each original edge is charged at most once.

For heap updates of type (1), the cost is thus $O(m \log n)$ time in expectation. For heap updates of type (2), there are at most $O(m)$ of these updates, and each costs $O(\log n)$ time for a total cost of $O(m \log n)$ time.

Finally, the remaining steps in the algorithm outside of the merge steps are $O(n)$ BestEdge queries, which each cost $O(\log n)$ time for a total cost of $O(n \log n)$ time. Thus, the overall time-complexity of the algorithm is $O(m \log n)$ in expectation, as desired.

□

D.3 Exact Unweighted Average-Linkage

First, we present again the two key subroutines that make up our exact unweighted average-linkage algorithm (Algorithms 5 and Algorithm 6).

Algorithm 5 FlipEdge(A, B)

Input: Edge oriented from active clusters A to B.
Output: Edge oriented from B to A.
1: $w = \mathcal{W}(A, B)$. [true weight of the edge]
2: Update the edge (A, B, w) in Heap(A).

Let $\hat{G}$ denote the directed graph induced by the active clusters, with edges oriented according to the edge-orientation $\mathcal{EO}$. We start by proving a lemma that helps prove that our exact unweighted average-linkage algorithm is correct.

Lemma D.6. *After a call to Algorithm 1, for each active cluster A, the weights of all edges directed towards A in $\hat{G}$ are set correctly in* Heap(A).

Algorithm 6 UpdateOrientation$(A, B, \mathcal{EO})$

Input: Active clusters A and B, dynamic orientation structure $\mathcal{EO}$.

1: $(A, B) = (\text{argmin}(d(A), d(B)), \text{argmax}(d(A), d(B))$.
2: For each $C \in N(A)$, delete (C, A) and insert (C, B) into the orientation data structure. Edge flips are handled using FlipEdge [Algorithm 5]
3: **for** each edge (B, C) oriented out of B **do**
4: $\quad w = \mathcal{W}(B, C)$. [true weight of the edge]
5: $\quad$ Update the edge (B, C, w) in Heap(C).

Proof. The proof is by induction. Consider the k-th merge between two clusters A and B, and assume that the claim holds before this merge. Assume without loss of generality that A is deactivated by this merge, and B remains active. Recall that the merge algorithm will also invoke Algorithm 6, which updates the orientation by remapping edges that are incident to the deactivated cluster from the maintained orientation $\mathcal{EO}$, and that Algorithm 5 is invoked each time the dynamic edge-orientation algorithm flips an edge.

For any edges that are flipped during the execution of Algorithm 6, the weights of these edges are correctly set in the heap of the cluster that the edge now points to (the *head* of this edge). The reason is that Algorithm 5 is invoked upon each edge flip and this algorithm computes the correct weight of the edge and updates the weight in the heap of the head of this edge. This accounts for all edges that are flipped by the orientation algorithm when deleting all (A, C) (undirected) edges and reinserting them as (B, C) edges.

The only remaining edges which may not have updated their out-neighbors are new edges incident to B that are oriented out of B. Thus, Algorithm 6 maps over all edges oriented out of B and sets the weight of these edges correctly in the heap of the head of this edge.

We have accounted for (i) all edges whose orientation flips due to the deletions and insertions in the merge and (ii) the new edges oriented out of B. Finally, by assumption, the remaining edges have their correct weights set in the heaps of the head of these edges, and thus all active clusters A have the correct weights for edges that point to them. □

Theorem D.7. *The exact average-linkage algorithm is correct and runs in $\tilde{O}(n\sqrt{m})$ time for arbitrary graphs.*

Proof. To show correctness, by Lemma D.6, we have that after the i-th merge, the state of each active cluster's heap is correct for all but the $O(\alpha_i)$ edges that are oriented out of this cluster. Since before performing a BestEdge computation, the algorithm maps over all $O(\alpha_i)$ of these edges and updates the weight of these edges in its heap to the correct weight, all of the edges in its heap have the correct weight, and thus the cluster selects the best-edge incident to it. The correctness of the overall algorithm can now by obtained by combining this proof with the existing proof for the correctness of the nearest-neighbor chain algorithm.

Next, we analyze the time-complexity of our algorithm. We first bound the extra cost incurred by maintaining the dynamic graph-orientation data structure $\mathcal{EO}$ over the course of the algorithm. Using the dynamic graph-orientation data structure of Henzinger et al. [HNW20], we obtain an amortized cost of $O(\log^2 n)$ for each edge insertion and deletion. The dynamic graph-orientation data structure is only updated and used during the Merge() and BestEdge() operations. Consider the i-th such operation, and let G_i be the graph induced by the current clustering at the time of this operation.

Using Lemma D.2, the total number of merge operations is at most $O(m \log n)$. For each of these operations, we have to perform an edge insertion and deletion, which could translate to a total $O(m \log^3 n)$ edge flips. Each edge flip also requires $O(\log n)$ time to update the weight of the edge in the heap of the new head of this edge. Thus the total cost of the edge insertions, deletions, and flips is $O(m \log^4 n)$ over the course of the entire algorithm.

The merge algorithm also processes the edges incident to the remaining active cluster, B. The cost for this step is $O(\alpha_i \log n)$, since there are $O(\alpha_i)$ edges oriented out of B and we pay $O(\log n)$ to perform a heap-update for each one. Since there are $n-1$ merges, the overall cost for this step is $O(\log n \sum_{i=1}^{n-1} \alpha_i)$ over the course of the entire algorithm.

Lastly, the cost for performing the BestEdge operation is $O(\alpha_i^*)$ where α_i^* is the current arboricity at the time of the i-th BestEdge operation. Note that there may be many BestEdge operations performed before a merge is performed. Since there are $2n-2$ BestEdge operations, the overall cost is $O(\sum_{i=1}^{2n-2} \alpha_i^*)$.

By bounding each $\alpha_i, 1 \leq i \leq n-1$ and $\alpha_j^*, 1 \leq j \leq 2n-2$ above as $\alpha_{\max} \leq \sqrt{m}$, the maximum arboricity of the graph over the entire sequence of merges, the overall time-complexity of the algorithm is

$$O(m \log^4 n + n \log n \cdot \alpha_{\max}) = \tilde{O}(n\sqrt{m}).$$

□

D.4 Approximate Unweighted Average-Linkage

We start by recalling the notion of approximation and the invariant used in our approximation algorithm. An *ϵ-close HAC algorithm* is an algorithm which only merges edges that have similarity at least $(1-\epsilon) \cdot \mathcal{W}_{\max}$ where $\mathcal{W}_{\max}$ is the largest weight currently in the graph [MLLL19].

The idea of our algorithm is to maintain an extra counter for each cluster which stores the size the cluster had at the last time that the algorithm updated *all* of the incident edges of the cluster. Call this variable the *staleness*, $\mathcal{S}(A)$, of a given cluster A. Recall that the size of a cluster $|A|$ is defined to be the number of initial (singleton) clusters that it contains. Our algorithm maintains the following invariant:

Invariant 1. *For any active cluster A, $|A| < (1+\epsilon)\mathcal{S}(A)$.*

Let the stored similarity of an edge (u, v) in the neighborhood be denoted $W_{\mathcal{S}}(u, v)$ and the true similarity of this edge be $W_{\mathcal{T}}(u, v)$. The next lemma bounds the maximum error an algorithm maintaining Invariant 1 can observe for an edge incident to an active cluster.

Lemma D.8. *Let $e = (U, V)$ be an edge in the neighborhood of an active cluster U. Then, $(1+\epsilon)^{-2} W_{\mathcal{S}}(U, V) \leq W_{\mathcal{T}}(U, V)$.*

Proof. There are two ways that the similarity of the (U, V) edge can change as the algorithm merges clusters:

1. By a merge which adds parallel edges to $\mathsf{Cut}(U, V)$ (e.g., if U merges with a cluster Z which is also connected to V, thereby increasing the total similarity of edges crossing the cut).
2. By a merge to $|U|$ or $|V|$ that does not affect the total similarity of edges going across $\mathsf{Cut}(U, V)$ (e.g., if U merges with a cluster Z that is not connected to V).

If a Type 1 update occurs, then the similarity of this edge will be set to the true value upon this update, since the algorithm will update the similarities of every edge in the intersection of the merge. Therefore, we can ignore these updates when trying to bound the maximum drift between the true and stored similarities. On the other hand, we could have many Type 2 updates occur.

In this case, the similarity of this edge will not be updated unless Invariant 1 becomes violated for either U or V. Therefore, we have the following upper bounds on the maximum size of U and V, namely that $|U| < (1+\epsilon)S(U)$ and $|V| < (1+\epsilon)S(V)$.

In the worst case, the stored similarity could have used $S(U)$ and $S(V)$ to normalize (since the similarity must have been updated when these stored similarities were set), and the true similarity could use $(1+\epsilon)S(U)$ and $(1+\epsilon)S(V)$. Since the sum term in the similarity equation doesn't change (since there are no Type 1 updates), we have that

$$W_{\mathcal{S}}(U,V) \leq \frac{1}{S(U)\cdot S(V)} \cdot \sum_{(u,v)\in \mathsf{Cut}(U,V)} w(u,v)$$

and therefore

$$W_{\mathcal{T}}(U,V) \geq \frac{1}{(1+\epsilon)^2\cdot S(U)\cdot S(V)} \cdot \sum_{(u,v)\in \mathsf{Cut}(U,V)}$$

Therefore, the true similarity is at most a $(1+\epsilon)^{-2}$ factor smaller than the stored similarity. □

Theorem D.9. *There is an ϵ-close HAC algorithm for the average-linkage measure that runs in $O(m\log^2 n)$ time.*

Proof. First we show that our algorithm is ϵ-close. Let δ be the closeness parameter used internally in the algorithm, which we will set shortly. For each (U,V) edge, since the simialrities in the algorithm only decrease, we have that $W_{\mathcal{T}}(U,V) \leq W_{\mathcal{S}}(U,V)$. Combining this fact with Lemma D.8, we have that

$$(1+\delta)^{-2}W_{\mathcal{S}}(U,V) \leq W_{\mathcal{T}}(U,V) \leq W_{\mathcal{S}}(U,V)$$

for all active edges U, V.

Next, consider a merge step in the algorithm which merges two clusters A, B. We have that $W_{\mathcal{S}}(A,B)$ is the largest stored similarity among any active cluster in the graph. We also have that $(1+\delta)^{-2}W_{\mathcal{S}}(A,B) \leq W_{\max}$ where $W_{\max}$ is the current maximum similarity in the graph, since otherwise the stored similarity corresponding to $W_{\max}$ would be larger than $W_{\mathcal{S}}(A,B)$ and thus (A,B) would not be the edge selected from the global heap, H. Therefore our approach yields a $(1-(1+\delta)^{-2})$-close algorithm, and by setting $\delta = \sqrt{1/(1-\epsilon)} - 1$ we obtain an ϵ-close algorithm.

Lastly, we show that the algorithm runs in $O(m\log^2 n)$ time for any given constant ϵ. Other than the extra work done to update stale clusters, the algorithm is identical to the heap-based algorithm from our framework. To bound the work done for stale clusters, observe that each cluster can become stale at most $O(\log_{1+\delta} n) = O(\log_{1+\epsilon} n)$ times, and performs $O(\log n)$ work per incident edge each time it becomes stale. Since each of the original m edges is associated with at most two active clusters at any point in the algorithm, the overall time-complexity of updating stale vertices is $O(m\log_{1+\epsilon} n \log n) = O(m\log^2 n)$ for any constant ϵ. Combining this with the time-complexity of the heap-based algorithm completes the proof. □

E Experimental Evaluation

Graph Data. We list information about graphs used in our experiments in Table 1. *com-DBLP* **(DB)** is a co-authorship network sourced from the DBLP computer science bibliography[1]. *YouTube*

[1]Source: https://snap.stanford.edu/data/com-DBLP.html.

Table 1: Graph inputs, including vertices and edges.

Graph Dataset	Num. Vertices	Num. Edges
com-DBLP **(DB)**	425,957	2,099,732
YouTube-Sym **(YT)**	1,138,499	5,980,886
Skitter-Sym **(SK)**	1,696,415	22,190,596
LiveJournal-Sym **(LJ)**	4,847,571	85,702,474
com-Orkut **(OK)**	3,072,627	234,370,166

Table 2: Adjusted Rand-Index (ARI) and Normalized Mutual Information (NMI) scores of our graph-based HAC implementations (columns 2–5) versus the HAC implementations from sklearn (columns 6–9). The scores are calculated by evaluating the clustering generated by each cut of the generated dendrogram to the ground-truth labels for each dataset. Our graph-based implementations run over an approximate k-NN graph with $k = 50$.

	Dataset	Single	Complete	WPGMA	Apx-Avg	Avg	Sk-Single	Sk-Complete	Sk-Avg	Sk-Ward
ARI	*iris*	0.702	0.462	0.605	**0.759**	**0.759**	0.714	0.642	**0.759**	0.731
	wine	0.297	0.286	0.317	0.331	0.331	0.297	**0.370**	0.351	0.368
	digits	0.661	0.133	0.500	0.876	**0.880**	0.661	0.478	0.689	0.812
	cancer	**0.561**	0.543	0.539	0.489	0.489	**0.561**	0.464	0.537	0.406
	faces	0.467	0.438	0.480	0.508	0.508	0.467	0.471	0.529	**0.608**
NMI	*iris*	0.733	0.641	0.733	**0.805**	**0.805**	0.761	0.722	**0.805**	0.770
	wine	0.410	0.388	0.387	0.427	0.427	0.417	**0.463**	0.448	0.448
	digits	0.772	0.572	0.713	0.900	**0.902**	0.772	0.711	0.838	0.868
	cancer	0.316	0.359	0.384	**0.460**	**0.460**	0.385	0.442	0.456	0.446
	faces	0.847	0.846	0.857	0.859	0.859	0.856	0.855	0.867	**0.871**

(YT) is a social-network formed by user-defined groups on the YouTube site[2]. *Skitter* **(SK)** is an internet topology graph generated from traceroutes[3]. *LiveJournal* **(LJ)** is a directed graph of the social network[4]. *com-Orkut* **(OK)** is an undirected graph of the Orkut social network[5]. These graphs are sourced from the SNAP dataset [LK14].

Another family of graphs that we consider are generated from point datasets by using an approximate k-NN graph construction. All of the point datasets that we use can be found in the sklearn.datasets package[6]. We note that the large real-world graphs that we study are not weighted, and so we set the similarity of an edge (u, v) to $\frac{1}{\log(d(u)+d(v))}$. We symmetrized all directed graph inputs considered in this paper.

References

[Ben82] Jean-Paul Benzécri. Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données*, 7(2):209–218, 1982.

[2]Source: https://snap.stanford.edu/data/com-Youtube.html.
[3]Source: https://snap.stanford.edu/data/as-Skitter.html.
[4]Source: https://snap.stanford.edu/data/soc-LiveJournal1.html.
[5]Source: https://snap.stanford.edu/data/com-Orkut.html.
[6]For more detailed information see https://scikit-learn.org/stable/datasets.html.

[BFS16] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2016.

[BT79] Mark R Brown and Robert E Tarjan. A fast merging algorithm. *Journal of the ACM (JACM)*, 26(2):211–226, 1979.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3), 1987.

[HNW20] Monika Henzinger, Stefan Neumann, and Andreas Wiese. Explicit and implicit dynamic coloring of graphs with bounded arboricity. *CoRR*, abs/2002.10142, 2020.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, 2014.

[MLLL19] Benjamin Moseley, Kefu Lu, Silvio Lattanzi, and Thomas Lavastida. A framework for parallelizing hierarchical clustering methods. In *ECML PKDD 2019*, 2019.

[MRS08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[SFB18] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. Pam: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.